

Efficient Pattern Search in Large, Partial-Order Data Sets

by

Matthew Nichols

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2008

© Matthew Nichols 2008

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The behaviour of a large, distributed system is inherently complex. One step towards making this behaviour more understandable to a user involves instrumenting the system and collecting data about its execution. We can model the data as traces (representing various sequential entities in the system such as single-threaded processes) that contain both events local to the trace and communication events involving another trace.

Visualizing this data provides a modest benefit to users as it makes basic interactions in the system clearer and, with some user effort, more complex interactions can be determined. Unfortunately, visualization by itself is not an adequate solution, especially for large numbers of events and complex interactions among traces. A search facility has the ability to make this event data more useful.

Work has been done previously on various frameworks and algorithms that could form the core of such a search facility; however, various shortcomings in the completeness of the frameworks and in the efficiency of the algorithms resulted in an inconsistent, incomplete, and inefficient solution.

This thesis takes steps to remedy this situation. We propose a provably-complete framework for determining precedence between sets of events and propose additions to a previous pattern-specification language so it can specify a wider variety of search patterns. We improve the efficiency of the existing search algorithm, and provide a new, more efficient, algorithm that processes a pattern in a fundamentally different way. Furthermore, the various proposed improvements have been implemented and are analysed empirically.

Acknowledgments

First, and most importantly, I would like to thank my supervisor Dr. David Taylor as he has provided significant intellectual, financial, and moral support for this endeavour. His attention to detail, sound judgement, and vast background knowledge in this area have been invaluable assets to me over the past six years.

I would also like to thank my readers from the University of Waterloo, Dr. Jay Black, Dr. Martin Karsten, and Dr. Todd Veldhuizen, and from IBM Corporation, Dr. Robert Wisniewski, for their time, helpful suggestions, and thoughtful insights into this work.

The Centre for Advanced Studies at the IBM Toronto Lab provided both generous financial support and many opportunities for me to present this work to a broader audience. I am very grateful to have had this arrangement with them. I would also like to acknowledge the financial and academic support of the University of Waterloo provided through various scholarships, awards, and teaching opportunities.

The students and faculty in the Shoshin Lab at the University of Waterloo, especially Dr. Paul Ward (and Dr. Jay Black and Dr. David Taylor as above), have provided a collegial atmosphere and were always very willing to discuss ideas and provide feedback. I would also like to thank the many faculty, staff, and fellow graduate students with whom I worked on various teaching-related undertakings.

Finally, on a personal note, I would like to thank those closest to me. Austin and my parents, David and Marianne, have been extremely supportive and have always encouraged me to excel in my academic endeavours.

Contents

List of Tables	viii
List of Figures	ix
1 Introduction	1
1.1 Thesis Contributions	2
1.2 Thesis Overview	3
2 Previous Work	5
2.1 Monitoring Distributed Systems	5
2.1.1 Distributed Systems and Applications	6
2.1.2 Application Monitoring	7
2.1.3 Event precedence	11
2.2 Pattern Languages	14
2.3 Compound Events	16
2.4 Predicate Detection	19
2.4.1 Partial-Order-Event-Based Predicate Detection	19
2.4.2 Parallels with Database Query Processing	21
3 A Framework for Comparing Sets of Partial-Order Events	22
3.1 Crossing and Overlapping Event Sets	23
3.2 Efficient Detection of Event Entanglement (and more)	28
3.3 Convex Closure and Its Validity	33
3.4 Additions to Pattern Language	37
3.5 Completeness of Operator Set	44
3.6 Evaluating Patterns	51

4	Convex-Closure Algorithm	52
4.1	Convex-Closure Theorems and Definitions	52
4.2	Existing Algorithm	54
4.2.1	Description of Existing Algorithm	54
4.2.2	Asymptotic Behaviour of Existing Algorithm	56
4.3	New Algorithm	58
4.3.1	Description of New Algorithm	58
4.3.2	Correctness of New Algorithm	60
4.3.3	Asymptotic Behaviour of New Algorithm	62
4.4	Additional Improvements	63
4.5	Combining Convex Closures	66
4.6	Test Setup	68
4.7	Performance Evaluation	71
5	Pattern Rewriting	89
5.1	Potential Benefits of Rewriting	90
5.1.1	Eliminating the Convex Closure	90
5.1.2	Reordering and Pruning Pattern Components	91
5.2	A Basis for Rewriting	92
5.3	Mechanics of Rewriting	100
5.4	Rewriting Examples	101
5.5	A Search Algorithm for Rewritten Patterns	104
5.6	Performance Evaluation	107
6	Optimizing Rewritten Patterns	110
6.1	Pruning	111
6.2	Static Reordering	114
6.2.1	Brute-Force Technique and Work Estimation	115
6.2.2	Least/Most-Successful-First Technique	118
6.2.3	Variable-Clustering Technique	119
6.3	Processing Universally-Quantified Variables	120
6.4	Performance Evaluation	121

7	Closing Remarks	129
7.1	Conclusions	129
7.2	Future Work	130
7.2.1	Dynamic Reordering of Rewritten Patterns	131
7.2.2	Mixed-Strategy Approach	131
7.2.3	Parallelization of Search Algorithm	132
7.2.4	Deeper Exploration of Reordering Metrics	133
7.2.5	Lower Bound on Convex-Closure Algorithm	133
7.2.6	Online Pattern Search	134
	References	134

List of Tables

3.1	Internal Representation of Events (TCP-Socket Environment) . . .	46
4.1	Pattern-Based Performance Analysis, Part 1	75
4.2	Pattern-Based Performance Analysis, Part 2	76
4.3	Random-Set Analysis for TCP Event Set, Part 1	79
4.4	Random-Set Analysis for TCP Event Set, Part 2	79
4.5	Random-Set Analysis for μ C++ Event Set, Part 1	79
4.6	Random-Set Analysis for μ C++ Event Set, Part 2	80
4.7	Random-Set Analysis for PVM Binary-Merge Event Set, Part 1 . .	80
4.8	Random-Set Analysis for PVM Binary-Merge Event Set, Part 2 . .	80
4.9	Random-Set Analysis for PVM Life Event Set, Part 1	80
4.10	Random-Set Analysis for PVM Life Event Set, Part 2	81
4.11	Closure-Combining Analysis for TCP Event Set	84
4.12	Closure-Combining Analysis for μ C++ Event Set	85
4.13	Closure-Combining Analysis for PVM Binary-Merge Event Set . . .	85
4.14	Closure-Combining Analysis for PVM Life Event Set	85
5.1	Performance Analysis for Pattern Rewriting	108
6.1	Pruning Performance Analysis	124
6.2	Reordering Performance Analysis	125
6.3	Pruning Performance Analysis	127
6.4	Full Performance Analysis	127

List of Figures

2.1	Architecture of POET	9
2.2	A Process-Time Diagram	10
2.3	A POET Screenshot	11
2.4	Fidge/Mattern Timestamps	12
2.5	Convexity of Event Sets	17
2.6	Pattern Parse Tree	20
3.1	Crossing Events	23
3.2	Overlapping Events	23
3.3	Non-entangled Event Sets	30
3.4	Entangled Event Sets	30
3.5	Original Event Set	33
3.6	Result of Convex-Closure Operation	34
3.7	Example of Universal Quantifier	39
3.8	Grammar for the New Pattern Language	40
3.9	TCP-Socket Environment	46
3.10	Example Pattern 1	48
3.11	Example Pattern 2	48
3.12	Example Pattern 3	49
3.13	Example Pattern 4	50
4.1	Existing Computation of the <i>front</i>	54
4.2	New Computation of the <i>front</i>	59
4.3	Starting a Second Instance of Eclipse	69
4.4	Importing a Partial-Order Data Set	70
4.5	μ C++ Patterns	73

4.6	PVM Patterns	74
4.7	Graph of Random-Set Analysis for TCP Event Set	81
4.8	Graph of Random-Set Analysis for μ C++ Event Set	82
4.9	Graph of Random-Set Analysis for PVM Binary-Merge Event Set	82
4.10	Graph of Random-Set Analysis for PVM Life Event Set	83
4.11	Graph of Closure-Combining Analysis for TCP Event Set	86
4.12	Graph of Closure-Combining Analysis for μ C++ Event Set	86
4.13	Graph of Closure-Combining Analysis for PVM Binary-Merge Event Set	87
4.14	Graph of Closure-Combining Analysis for PVM Life Event Set	87
5.1	Pattern Parse Tree (Example 1)	101
5.2	Pattern Parse Tree (Example 2)	102
6.1	Inefficiencies Added, Example 1	123
6.2	Inefficiencies Added, Example 2	124
6.3	Test Cases 9, 9.1, and 9.2	125

Chapter 1

Introduction

AS COMPUTER programs and systems increase in size and complexity, we require more powerful and elaborate tools to help us manage this complexity. Distributed and multi-CPU systems are becoming more mainstream as increasing the clock speed of individual processors becomes increasingly difficult [52]. Thus, there is a strong need for such tools. Collecting information about a system at runtime and then later examining this information is a good starting point and we can then use this collected data to assist us for purposes of testing or debugging a system [13, 17, 62], visualizing its execution [15, 20, 30, 34, 36, 38], better understanding its behaviour [35], making it more dependable, or improving its performance [45, 48].

Unfortunately, in many cases, the amount of data collected is very large and many existing techniques require a human user to examine a visualization of the data manually for points of interest. This can be time-consuming, tedious, and often impractical.

One way to make this data more manageable and useful is through the use of a search facility that allows points of interest to be located quickly and provides for automatic verification of various properties of the collected data. Ideally, a search

facility would be very efficient, allow expressive search patterns to be specified, and provide consistent, well-defined results. Some previous work has been done on the search itself [9, 24, 33, 46, 47, 50, 58, 64] as well as some of the fundamental framework behind it [7, 13, 36, 51], but these goals have not yet been achieved.

The primary task of this thesis is to work towards these goals. We would like to provide a complete and well-defined framework for evaluating precedence between event sets, increase the expressiveness of the language through which searches are specified, and improve the runtime of the search algorithm through a variety of means.

1.1 Thesis Contributions

Many of the contributions of this thesis directly improve on previous work. Some work was done previously on developing an appropriate framework to evaluate relationships between sets of partial-order events [7, 33, 58], but this work did not provide a complete solution. A pattern language allowing the specification of patterns based on partial-order operations was also previously developed [58, 64], but was not expressive enough to search for some important types of patterns. Finally, some work was done that used convex events as part of the pattern search [58, 63]. Convex events are event sets where if a and b are events in the set, then any event c where $a \rightarrow c \rightarrow b$ is also in the set. Using convex events allows relationships between event sets to be determined more easily.

Specifically, this thesis provides the following contributions towards the goals mentioned earlier.

1. We provide a provably-complete framework for evaluating precedence between event sets. Frameworks used in previous work were incomplete and did not

- fully address the unique issues of evaluating event sets versus single events.
2. We add a variety of general features to the language that is used to specify search patterns and we demonstrate the increased expressiveness that these features provide. A full grammar for the language is provided.
 3. We greatly improve the runtime of the algorithm that builds convex events and we show how any two convex events can be combined without extracting the individual events from each convex event and completely rebuilding the convex event.
 4. We demonstrate how a search pattern can be rewritten into a less complex and more flexible form (in effect, we compile the search pattern). This rewritten form of the pattern can be evaluated more quickly.
 5. We show how various optimizations to the rewritten pattern and the processing algorithm can further improve the runtime of the pattern search.

1.2 Thesis Overview

The main contributions of this thesis are presented in four chapters.

Chapter 3 begins by developing a sound framework for comparing precedence between event sets and then shows how such comparisons can be done efficiently. We then present arguments that determining precedence between two convex events is equivalent to comparing the two original non-convex events. Finally, we propose additions to a previously-developed pattern language and demonstrate how these improvements increase its expressiveness.

Chapter 4 focuses on improving the runtime of the convex-closure algorithm. This algorithm is responsible for taking a set of events and returning an equivalent

convex set. We also show how we can combine two convex closures into a single closure by leveraging the previous work done in computing the original two closures. During the pattern search, it is often necessary to combine two closures, so this work has direct relevance to pattern search. Finally, we present a variety of empirical analyses of the improvements.

Chapter 5 presents a methodology for rewriting patterns based on the framework of Chapter 3. When we evaluate rewritten patterns, we do not need to perform convex-closure operations and this results in significant improvements to the runtime of the pattern search. Again, we empirically evaluate the improvements of evaluating patterns rewritten in this manner.

Chapter 6 builds on the work done in Chapter 5. Rewriting a pattern, by itself, improves the runtime of the search algorithm; however, it also allows for more flexibility when evaluating a pattern. We propose various ways to improve the runtime of the pattern search by reorganizing the rewritten patterns and modifying how the search algorithm processes certain pattern elements. We again use empirical analyses to evaluate the proposed improvements.

Chapter 2

Previous Work

THERE IS a substantial amount of background and related work in the area of pattern search and processing of partial-order event data. This chapter will first provide some background on distributed systems, application monitoring, and event precedence. We then explore previous work on pattern languages in this domain and look at problems surrounding the evaluation of precedence between two sets of events. Additionally, we look at how predicate detection is done currently on partial-order event data and how database query processing is related, at a high level, to our problem.

2.1 Monitoring Distributed Systems

Pattern search depends on the ability to monitor and collect information about the execution of a distributed system. This section provides information about how this monitoring and collection of information is accomplished and describes several intrinsic challenges.

2.1.1 Distributed Systems and Applications

A distributed system is one in which components located on multiple computers connected by a network cannot communicate through shared main storage and therefore typically coordinate their actions by passing messages [16]. Usually, the communication network used for passing messages has significantly lower bandwidth and higher latency than the internal communication bus of each individual machine. A distributed application is one consisting of a set of related, interacting processes running on a distributed system [24]. These processes typically run on multiple machines, are designed to communicate with each other using a network, and work together to provide the necessary functions of the application.

As raising the clock speed of processors becomes impractical for cost or other reasons, it is becoming increasingly attractive, from a hardware perspective, to create more multi-processor and distributed systems with budget-priced CPUs [18, 28]. Major manufacturers of CPUs are incorporating a “multi-core” design into single processors that allows the single processor to run as if it were multiple independent CPUs. Furthermore, many supercomputers, such as Blue Gene [1], integrate tens of thousands of multi-core processors. These trends will likely further motivate the development of and need for understanding of not only multi-threaded applications but also distributed applications. In turn, this emphasis on distributed and multi-threaded applications provides a greater impetus for the improvement of tools that can help develop such applications.

Usually, distributed applications are significantly more complex to construct and maintain than stand-alone applications. There are several reasons for this [23]. Primarily, it is because distributed applications are, by nature, non-deterministic. Absent any communication and synchronization between processes, there is no way to guarantee in which order the various steps of each process will execute, unless

a replay facility [14, 40, 49] is used. The only certainty is that a single process (or thread, when dealing with a multi-threaded process) will execute its steps in a predictable order. The result is that there can be little consistency in the actions of an imperfect distributed application from one execution to the next. This makes the job of debugging a distributed application significantly more challenging. Errors that appear in one run of the application are often not repeated in subsequent runs.

Another challenge of distributed applications is that there is no well-defined global state accessible to any individual machine. Tools such as debuggers that need access to information across all of the processes, such as the state of variables and program counters, are much more challenging to construct. In addition, whenever a process requires information held by another process, it needs to communicate with that process to retrieve it. This leads to additional complications involving staleness of data.

A third challenge of distributed applications is the lack of a global clock. Each machine has its own time source, so in the absence of very specialized hardware, there is no way to determine an exact real-time ordering on events that happen across multiple machines. This challenge is one of the driving reasons behind the use of a partial order to represent the ordering of events in a distributed system. A partial-order set is a pair (X, P) where X is a finite set and P is a reflexive, antisymmetric, and transitive binary relation on X [39]. Using such a representation allows us to minimize the uncertainties involved with system clocks.

2.1.2 Application Monitoring

Application monitoring involves collecting and processing data about the execution of an application [37, 41]. Typically, this data is then made available to a user and is used for a wide variety of purposes including debugging, performance analysis,

and live monitoring of an application for error conditions. In some cases, such as live monitoring, data is processed and analyzed as the monitored application is running (on-line). In other cases, data is collected and then analyzed later (off-line). This thesis will only examine an off-line approach; however, adapting this work to approximate on-line methods is feasible.

In most cases, the collection of data is achieved by adding extra code (instrumentation) to an application. Typically, either the underlying system or the libraries used by the application are modified. The purpose of the instrumentation is to record the appropriate information as an application executes and then, in most cases, transmit that information elsewhere for processing and analysis. Recording every piece of data about the execution of an application is rarely practical; thus, the instrumentation must focus on specific events or metrics or provide a mechanism that allows a user to choose which will be collected. Typically, the more data that is collected, the greater impact the data collection will have on the behaviour of the application and the further the application will stray from its true, non-instrumented behaviour.

The work in this thesis is built on top of the various techniques and algorithms in an existing tool, the Partial-Order Event Tracer (POET) [37, 56]. This tool allows a user to instrument a distributed application and collect information about it. POET itself is a distributed application and its architecture is shown in Figure 2.1 [56]. POET is target-system independent, which means that it can collect data from a wide variety of execution (target) environments with a minimum amount of effort in most cases. The original tool was written in C for UNIX-based systems; however, a large part of its functionality has been ported to Java.

The Java port of POET is built using the Eclipse SDK [2] and relies on a portion of the functionality of Eclipse at runtime. In particular, the user interface is designed as a plugin for Eclipse. The plugin allows a user to import a data set

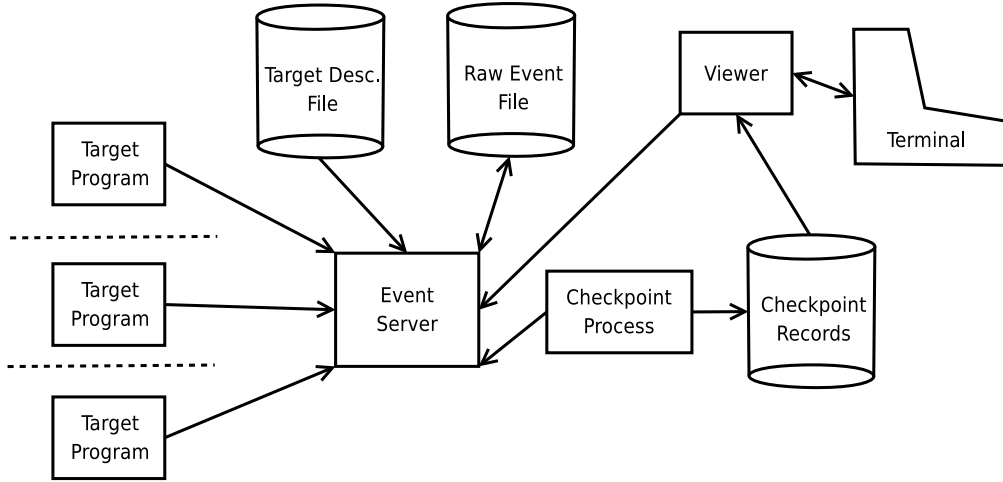


Figure 2.1: Architecture of POET

collected by the original C version of POET, displays the data set to the user in a viewer, and allows the user to search for various event patterns within the data set.

The core information stored by POET is a set of events and the partial-order relationships among those events. The occurrence of a POET event indicates that one of a predefined set of important actions has occurred in an application that is part of the system. This set of important actions is entirely dependent on the target environment being used. For example, a TCP-socket target environment would define the important actions to be `bind`, `send`, and `recv` socket calls, among others. As the events are collected, they are grouped by trace, typically one trace per process. The relationship between events is recorded as well. For example, in one target environment it might be specified that the event of sending of a message is paired with the event of receiving the message.

To present this information to the user, POET contains a graphical viewer. The viewer presents the traces as horizontal lines, where time flows from left to right, and relationships between pairs of events that belong to different traces are drawn as vertical or diagonal lines that connect the two events. Figure 2.2 illustrates the

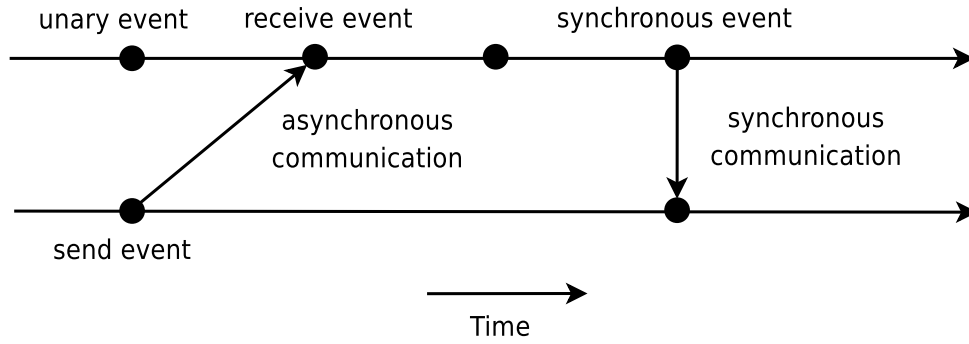


Figure 2.2: A Process-Time Diagram

style of process-time diagram that the POET display uses. The diagram shows how three different types of fundamental events are displayed: unary events, which stand alone on a trace; synchronous communication events, indicated by a vertical line; and asynchronous communication events indicated by a diagonal line.

In most cases, the set of events will extend far beyond the boundaries of the screen. Because these events are in a partial order, using a standard horizontal scroll bar to move to events located off screen would be inappropriate. This type of scroll would mislead the user into thinking that there is an absolute ordering between all events as the events would always be in the same relative positions. Two concurrent events, for example, might always appear as though one happened before the other.

To avoid presenting a misleading view of the events, a partial-order scroll algorithm was devised [57]. This algorithm allows a user to scroll through a single trace in a predictable order, and then adjusts the surrounding events in an appropriate manner. One of the goals in adjusting surrounding events is that they should be shifted as little as possible. This means, for example, that if there is no precedence relationship between the two traces (i.e., all events on one trace are concurrent with all events on the other trace), when one trace is being scrolled, the other trace will

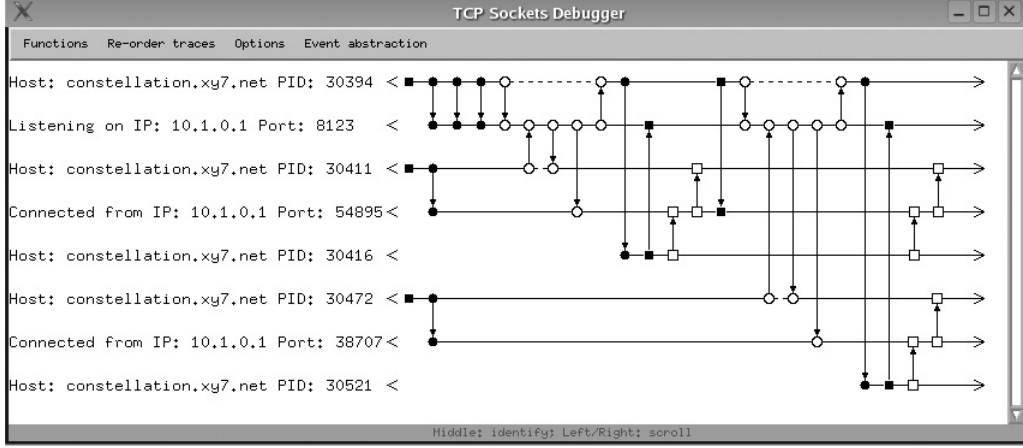


Figure 2.3: A POET Screenshot

not change. Figure 2.3 shows a screenshot of POET with data collected from a TCP-socket target environment [45]. There are additional events to the right of the display window that can be scrolled to.

POET allows the user to manipulate the view in other ways as well. The ordering of traces can be changed and multiple traces/events can be collapsed into a single trace/event.

2.1.3 Event precedence

In order to efficiently determine precedence relationships between single events, POET uses Fidge/Mattern vector timestamps [21, 22, 42]. An event's timestamp can be used to quickly determine the precedence relationship that exists between two single events and also determine an event's greatest predecessors on each trace.

Definition 2.1. The greatest predecessor of an event, a , on trace p denoted $GP(a, p)$ is the single-element set containing the most-recent event, $\{e\}$, on trace p that happens before a , or the empty set, $\{\}$, if no such event exists. Happens-before relationships are the fundamental relationships of a partial order.

Definition 2.2. The least successor of an event, a , on trace p denoted $LS(a, p)$ is

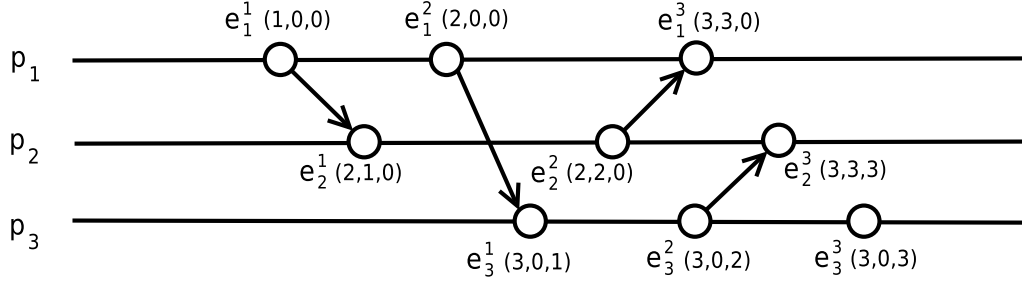


Figure 2.4: Fidge/Mattern Timestamps

the single-element set containing least-recent event, $\{e\}$, on trace p that happens after a , or the empty set, $\{\}$, if no such event exists.

Definition 2.3. Where a pair of events indicate a synchronous or asynchronous communication between two traces, we say that one event is the *partner* of the other.

Definition 2.4. The set of immediate predecessors of an event, a , where a is located on trace p , is the union of $GP(a, p)$ and the partner of a , if the partner exists and happens before a , or $GP(a, p)$ otherwise.

Definition 2.5. The set of immediate successors of an event, a , where a is located on trace p , is the union of $LS(a, p)$ and the partner of a , if the partner exists and happens after a , or $LS(a, p)$ otherwise.

Knowledge about precedence among events is critical when searching for event patterns in partial-order data. Figure 2.4 provides an example showing the first three events on each of the three traces in the event set. The trace number and event number of each event are shown, respectively, as subscript and superscript integers and the timestamp for each event is shown in brackets. Intuitively, an event's timestamp describes the number of events on each trace that precede it (as long as we first subtract a value of 1 from each non-zero integer in the timestamp).

For example, two events from trace 1, no events from trace 2, and two events from trace 3 event precede e_3^3 , which has a timestamp of (3,0,3).

In POET, these timestamps are pre-computed upon loading the data set. As the timestamps are pre-computed, information about the state of the timestamping process is periodically stored such that the timestamp for a given event can later be retrieved quickly. Each timestamp is an array of n integers, where n is the total number of traces in the complete set of events. With timestamps, we can quickly determine, for two primitive events a and b , the precedence relationship between the events. The only possible relationships for these two primitive events are either that a happens before b , b happens before a , a is concurrent with b , or a and b are equal. We can check if either of the first two relationships holds with at most two integer comparisons. If neither of these holds, we can determine which of the last two relationships holds with at most two additional comparisons as we need to compare the trace numbers and event numbers for equality. For example, if event a occurs on process p_a and has vector timestamp t_a and event b occurs on process p_b and has vector timestamp t_b , then to determine if a happens before b , we simply check if $t_a[p_a] < t_b[p_a]$. If the inequality is true, then a happens before b ; otherwise, it does not.

To illustrate this, we can further examine the events in Figure 2.4. If we want to determine if e_1^1 happens before e_3^3 , we would check whether $t_{e_1^1}[p_{e_1^1}] = t_{e_1^1}[1] = 1 < 3 = t_{e_3^3}[1] = t_{e_3^3}[p_{e_1^1}]$ (assuming the timestamp vector is indexed starting at 1). Since $1 < 3$, e_1^1 happens before e_3^3 . If we wanted to determine if e_2^1 is concurrent with e_3^3 , we would check both whether $t_{e_2^1}[p_{e_2^1}] = t_{e_2^1}[2] = 1 \not< 0 = t_{e_3^3}[2] = t_{e_3^3}[p_{e_2^1}]$ and $t_{e_3^3}[p_{e_3^3}] = t_{e_3^3}[3] = 3 \not< 0 = t_{e_2^1}[3] = t_{e_2^1}[p_{e_3^3}]$ (we need to verify that e_2^1 does not happen before e_3^3 and vice versa). Since $1 \not< 0$ and $3 \not< 0$ (and e_2^1 and e_3^3 are not the same event) then e_2^1 is concurrent with e_3^3 . In Chapter 3 we will explore how precedence can be determined between two *sets* of events.

Although the number of entries in each Fidge-Mattern timestamp is equal to the number of traces in the data set, previous work on scalability issues for vector timestamps [26, 59, 60, 61, 62] has looked at ways to reduce this requirement. Unfortunately, finding the minimum vector length is NP-hard [31]. Known schemes producing short vectors are quite complex and in some cases can only compute timestamps offline.

2.2 Pattern Languages

In order to search for a pattern, we first need a way to specify one. To accomplish this, a predicate language (pattern language) needs to be defined. A predicate language defines what constitutes a valid predicate and how to interpret predicates. A pattern-search algorithm then uses this information to return matches that are consistent with the given predicate.

In the context of distributed systems, several different predicate languages have been proposed [8, 24, 29, 33, 43]. The creators of each language had different purposes in mind when creating their languages, so each one defines different operators that narrow the purpose of the language and help the creators achieve their goals in a direct manner. Some are designed for event data that is ordered based on the real times of events, whereas others assume a partial ordering of events.

Examining many predicate languages shows that there are core elements that are desirable for inclusion in a partial-order-based predicate language. First, there must be a way to restrict individual events based on the attributes of that event. For example, if events have a “type” attribute, we may want to limit our search to *send* events ($e.type = \text{send}$). It is convenient to define an event class (e.g., \mathcal{A}) which can be assigned a set of event-attribute restrictions. We can then use the class identifier in patterns.

Second, there must be a way to restrict pairs of events based on their partial-order relationships. A set of operators can be defined to facilitate this. Two of the most basic operators are *happens before* (\rightarrow) and *concurrent* (\parallel); other, more elaborate, operators have been proposed as well.

Finally, there must be a way to link together the above components. At a basic level, there can be conjunctive and disjunctive operators (\wedge and \vee). Components can also be linked together using multiple operators and brackets. For example, evaluating $\mathcal{A} \rightarrow (\mathcal{B} \parallel \mathcal{C})$ would involve first finding an event that satisfies class \mathcal{B} and another that satisfies class \mathcal{C} and then determining if the two events are concurrent. If so, finding an event that matches \mathcal{A} is the next step and we must then check that \mathcal{A} happens before the pair of events matching the bracketed expression. This is more complicated than it appears, since the bracketed expression is not a primitive event. We discuss this issue further below, but Chapter 3 presents a solution to this complication.

Parts of the pattern language specified in Chapter 3 are built upon ideas presented in two different predicate-language specifications created by Jaekl [33] and Fox [24]. (In turn, these predicate-language specifications were also built upon the ideas from others' languages.) Jaekl's predicate language uses operators based on the work of Haban and Weigel [29], and has a fixed triple of attributes for each event class: the name of the process the event is in, the type of the event, and a comment. Each of these three fields can be specified explicitly or can be represented as a regular expression that would potentially match multiple different strings.

In addition to the *happens-before* and *concurrent* operators, Jaekl's language includes a *limited* operator ($\xrightarrow{\text{limited}}$). This operator is similar to the *happens-before* operator, but involves an additional event. For example, if our pattern specifies that \mathcal{A} happens before \mathcal{B} , but is limited by \mathcal{C} ($A \xrightarrow{\mathcal{C}} B$), it must hold that the match to \mathcal{A} happens before the match to \mathcal{B} , but cannot be the case that both the

match to \mathcal{A} happens before the match to \mathcal{C} and the match to \mathcal{C} happens before the match to \mathcal{B} . Jaekl also adds a send-receive-pair operator, which allows the user to specify that a given event class must be partnered with another specific event class. The existing implementation of pattern search in Java-based POET uses Jaekl’s predicate language.

Fox extended Jaekl’s predicate language to make it more powerful. He added variable binding, which allows a pattern to use variables, declared as belonging to a specific class, in place of the name of the event class itself. Using variables allows the pattern to refer back to a previously-matched event. For example, if we declare variable $\$a$ as belonging to class A , then the pattern $\mathcal{B} \rightarrow \$a \wedge \$a \rightarrow \mathcal{C}$ would be equivalent to $\mathcal{B} \rightarrow \mathcal{A} \rightarrow \mathcal{C}$ (but not equivalent to $\mathcal{B} \rightarrow \mathcal{A} \wedge \mathcal{A} \rightarrow \mathcal{C}$ because each \mathcal{A} could be bound to a different event). Fox also added additional operators that allow a user to place limits on how far to search into the future for a particular match. Fox’s focus was on on-line event-predicate detection (versus our focus on an off-line approach) so these limits were of particular value to him. The variable binding suggested by Fox is one of the important features added to our new pattern language in Chapter 3.

2.3 Compound Events

Determining precedence between two individual primitive events is well-defined; however, determining precedence when a compound event (an event that is composed of multiple primitive events) is involved is more challenging. Precedence for compound events has been defined in at least two different ways. The first definition states that a compound event happens before another compound event if and only if *all* of the primitive events in one compound event happen before *all* of the primitive events in the other. The second states that a compound event happens before

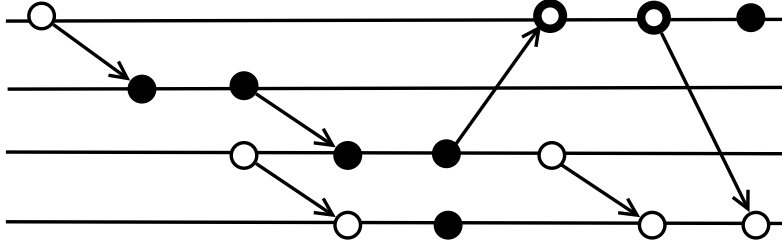


Figure 2.5: Convexity of Event Sets

another compound event if and only if there exists a happens-before relationship between *at least one* of the primitive events in each compound event.

The first of these definitions was favoured by Jaekl [33], whereas the second was favoured by Kunz and Xie [37, 64]. Other work suggests both may be useful but does not describe how they could be used simultaneously [7]. The first definition results in maintaining a valid partial-order relationship between events (either compound or primitive) with no extra effort, whereas the second requires that restrictions be placed on which events can form a compound event and it also lacks transitivity (i.e., it is not the case that $\mathcal{A} \rightarrow \mathcal{B} \wedge \mathcal{B} \rightarrow \mathcal{C} \Rightarrow \mathcal{A} \rightarrow \mathcal{C}$). The second definition contradicts the partial-order relationship because it is possible that, when it is used, a compound event happens simultaneously before and after another primitive or compound event. The problem is avoided when using the first definition as it requires that the precedence relation holds for all events in the compound event, not just a single one.

Given that, using the first definition appears more convenient; however, it was found that the second definition is more intuitive [36, 37] and, as long as restrictions are placed on which events can form a compound event, the problem mentioned above is avoided. To describe these restrictions, the notion of a *convex* event was developed. A compound event made up of a set of primitive events, E , is *convex* if and only if $\forall x, y \in E : x \rightarrow z \rightarrow y \Rightarrow z \in E$. The solid black events in Figure 2.5

do not form a convex event, but if we include both the solid black events and the events with a thick outline, then the event set is convex. If only *convex* compound events are used then the definition of a partial order will not be contradicted, in most cases, except for transitivity. It should be noted, however, that convex events are not a complete solution to determining precedence between compound events. It is possible to create two disjoint convex events that are positioned in such a way that they are not concurrent and each happens before the other. Earlier work avoids comparisons of these types of convex events; however, we develop a solution to this problem in Chapter 3.

When using the second definition, a given compound event may need to be expanded to make it convex so that the partial-order properties are not violated. To accomplish this, a convex-closure algorithm was created [58, 64] that can take any arbitrary set of events and transform it into a convex event. Convex closures are routinely performed as a core component of the current search algorithm. They are a necessary part of the algorithm as, without them, we would be unable to make meaningful precedence comparisons between two compound events and thus would be unable to properly search for anything but simple patterns. The closure algorithm takes a compound event, E , represented as a set of primitive events as input, and then returns a new, minimal set of events, E' , such that $\forall x, y \in E : x \rightarrow z \rightarrow y \Rightarrow x, y, z \in E'$. The result is that no primitive event, other than those in E will happen both before and after E , i.e., E is convex (although, as mentioned previously, another convex event could still happen both before and after E in some situations). Since the existing pattern search only allows comparisons between disjoint event sets, we can be certain that any primitive event in a closure can never be compared against that closure for precedence.

It is also useful to note that a convex event can be represented by storing only two primitive events per trace, rather than all primitive events in the convex event.

In other words, a convex event can be represented with two vectors of size n , where n is the number of traces in the complete event set. One vector holds the oldest primitive events in the convex event, one per trace (*front* vector), while the other vector holds the newest primitive events (*back* vector). The convex-closure algorithm, as its output, returns a *front* and a *back* vector.

2.4 Predicate Detection

This section will present some background information on predicate detection in the context of partial-order event data. Also, we will examine briefly some parallels between database query processing and predicate detection.

2.4.1 Partial-Order-Event-Based Predicate Detection

In recent work on predicate detection (pattern search) in partial-order data [33, 58, 64], the search itself is seen as a constraint-satisfaction programming problem (CSP). Each event pattern can be seen as a set of variables and a set of restrictions on their respective values. It seems likely that, in the contexts of debugging and program understanding, a search that is able to discover all possible matches, not just a random subset, is desirable, so a systematic backtracking search is the best choice. Improving the runtime of the backtracking pattern search is one of the primary goals of this thesis.

The backtracking search is similar to other such searches; however, in our context, when a match is found that is a compound event, the compound event must be converted to a convex event (by using a convex-closure algorithm). For example, when searching for $\mathcal{A} \parallel (\mathcal{B} \rightarrow \mathcal{C})$, the events that match \mathcal{B} and \mathcal{C} must be converted to a convex event before testing whether they are concurrent with \mathcal{A} .

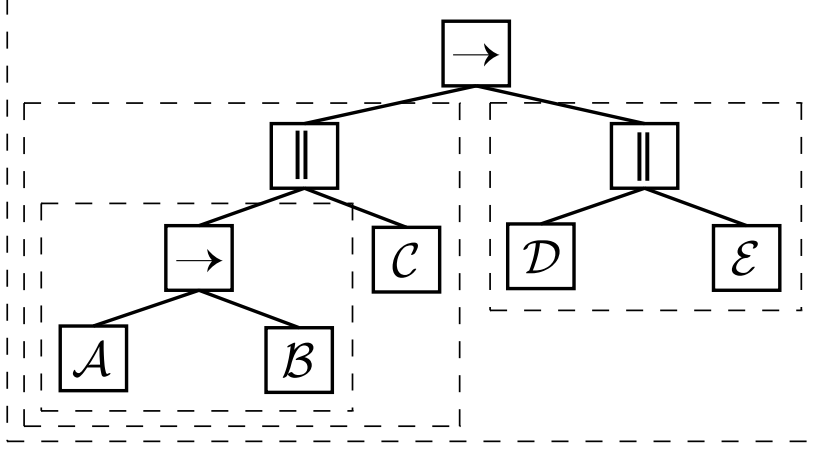


Figure 2.6: Pattern Parse Tree

Every time we backtrack and a new match for \mathcal{B} or \mathcal{C} is found, the convex closure of the two primitive events matching \mathcal{B} and \mathcal{C} must be re-computed. Because the convex-closure algorithm is used at each step in the backtracking search, significant improvements in the speed of that algorithm will make a noticeable difference in the overall speed of the search algorithm.

Figure 2.6 illustrates the parse tree for the pattern $((\mathcal{A} \rightarrow \mathcal{B}) \parallel \mathcal{C}) \rightarrow (\mathcal{D} \parallel \mathcal{E})$ and the dotted-line rectangles illustrate the convex closures that would be computed on a successful match. The original predicate-detection algorithm would begin by finding matches for \mathcal{A} and \mathcal{B} where $\mathcal{A} \rightarrow \mathcal{B}$. Then it would compute the convex closure of the two primitive events that matched \mathcal{A} and \mathcal{B} . Next, it would attempt to find a match for \mathcal{C} that is concurrent with the convex closure of the two primitive events that matched \mathcal{A} and \mathcal{B} . If successful, it would then try to find matches for \mathcal{D} and \mathcal{E} where $\mathcal{D} \parallel \mathcal{E}$ and compute the convex closure of the two primitive events that matched \mathcal{D} and \mathcal{E} . Finally, the algorithm would check that the convex closure of the three primitive events that matched \mathcal{A} , \mathcal{B} , and \mathcal{C} happens before the convex closure of the two primitive events that matched \mathcal{D} and \mathcal{E} . At any step in the process, if a given operator is not satisfied, the algorithm backtracks one step and attempts to find a new match for the one of the operands of the previous operator.

Work has also been done on predicate detection in distributed systems using a state-space approach [6, 19, 25, 27, 44, 54, 65]. Although some of the general goals of event-based and state-space-based approaches are similar, the two domains require vastly different approaches as they consume different types of input data. Additionally, the outputs of the surveyed state-space approaches are “yes” or “no” answers; whereas, our goal is to provide a full list of matching sets of events.

2.4.2 Parallels with Database Query Processing

At a high level, database query processing [12, 32] has similarities with event-based pattern search. Both typically operate on large data sets where the various pieces of data are correlated in some manner: in a typical database, correlation is often based on equivalence between values in one or more columns; whereas, in a partial-order context, the correlation is based on precedence relationships between events or sets of events. Additionally, in each domain, a query (possibly complex and hierarchical) is provided as input and some subset of the data is output as a match to the query.

We can use ideas from database query processing directly in some parts of the pattern search. For example, we store the event data set in a database and use database queries to extract a set of events satisfying a certain event class (based on the attributes of the events). Unfortunately, once we begin evaluating precedence relationships, there is no direct way of using a database to do so. We can, however, adapt some of the ideas used in processing and optimizing database queries (such as compiling or rewriting the query and then optimizing the ordering of operations in the query based on various data metrics) to develop techniques that can help us improve the processing speed of partial-order-event-based queries. Chapters 5 and 6 will present our techniques in detail.

Chapter 3

A Framework for Comparing Sets of Partial-Order Events

A WELL-DEFINED framework exists for comparing single events within a partial-order data set. While frameworks for comparing two *sets* of events exist [7, 33, 58], they have some shortcomings, as discussed previously. This chapter will present a new framework that resolves many of the inadequacies of the previous frameworks.

We first define a new operator to capture situations where two event sets cross (Figure 3.1) or overlap (Figure 3.2). In either case, we say that the two event sets are *entangled*. We then redefine the happens-before operator in a more precise way such that it does not match event sets that are entangled. Determining whether two event sets are entangled is an important operation. We present a well-defined, efficient algorithm to compute this.

We also make additions and improvements to the pattern language to increase the complexity and variety of patterns that we can search for. In particular, we add variable binding to the pattern language and fully define the meaning of Boolean

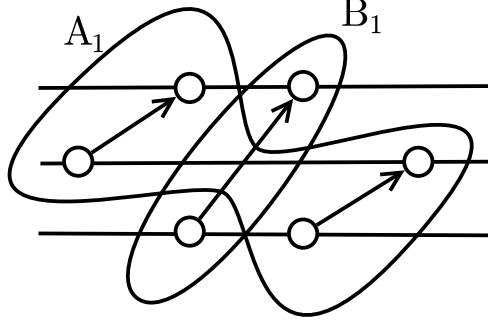


Figure 3.1: Crossing Events

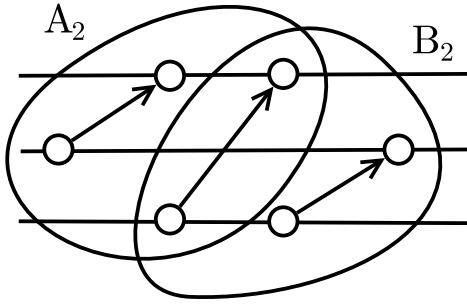


Figure 3.2: Overlapping Events

operators “and” (\wedge), “or” (\vee), and “not” ($!$). The new entanglement operator also increases the variety of searchable patterns.

We argue that performing a convex closure on each event set prior to comparing it to other sets or single events is a valid and logical mechanism that does not distort our pattern search. We then conclude by giving some examples of useful patterns that can be matched with the new framework.

3.1 Crossing and Overlapping Event Sets

In this section, we provide background information on the problems faced when encountering crossing and overlapping event sets. We then provide a solution that involves modifying existing definitions and introducing a new operator. These mod-

ified definitions and the new operator form the base of our new framework for comparing sets of partial-order events.

The original definition of *happens before* for event sets (3.1) requires only that a single event from each event set have a precedence relationship. As we will see shortly, it was necessary to make many assumptions (such as the event sets being disjoint) for this definition to be useful.

Definition 3.1. $A \rightarrow B \Leftrightarrow \exists x \in A, y \in B \mid x \rightarrow y$ (Original Happens-Before)

Previous work, as mentioned, has shown that this *any-any* definition is more intuitive than the *all-all* definition, which required that all events in the first set happen before all events in the second set. Although the *any-any* definition is more intuitive, one well-known problem with this definition is that it is possible to have two event sets, A and B , where both $A \rightarrow B$ and $B \rightarrow A$. In this configuration, we say that A *crosses* B (3.4). Because we would like the behaviour and properties of partial-order operators between sets of events to mimic those of operators between single events as much as possible, we would like to find a way to avoid this kind of contradiction. Another shortcoming of the previous work is the treatment of overlapping event sets.

Definition 3.2. $A \text{ overlaps } B \Leftrightarrow A \cap B \neq \emptyset$ (Overlap)

Definition 3.3. $A \text{ is disjoint from } B \Leftrightarrow A \cap B = \emptyset$ (Disjoint)

Definition 3.4. $A \text{ crosses } B \Leftrightarrow \exists x_0 \in A, y_0 \in B \mid x_0 \rightarrow y_0 \wedge \exists x_1 \in A, y_1 \in B \mid y_1 \rightarrow x_1 \wedge A \text{ is disjoint from } B$ (Cross)

Previous work assumed that event sets did not overlap (i.e., it assumed that they were disjoint) and so the framework did not define any means by which to discover overlapping event sets and did not integrate this concept into the pattern language.

Our solution to these problems is two-fold: add a new operator (\leftrightarrow) to recognize *entanglement* (3.5) of two event sets and modify the event-set definition of *happens before*.

The first step is to add the *entanglement* operator that allows us to recognize entangled event sets within the new framework. We have defined entanglement (3.5) to include both overlapping and crossing event sets as the characteristics of these two categories of event-set relationships are similar. It would be possible to add yet another operator to distinguish between these two types of relationships in the future.

Definition 3.5. $A \leftrightarrow B \Leftrightarrow A \text{ crosses } B \vee A \text{ overlaps } B$ (Entanglement)

Definition 3.6. $A \nleftrightarrow B \Leftrightarrow \neg(A \leftrightarrow B)$ (i.e., A does not cross $B \wedge A$ is disjoint from B) (Non-entanglement)

As a second step, we need to modify the *happens before* definition to be more restrictive such that two entangled sets of events do not satisfy it. To accomplish this, we add an extra restriction to the definition (3.7) that requires that no event from the second set happen before any event from the first set. Because we now include overlapping event sets in our framework, we also need to restrict the *happens before* operator such that it does not match two overlapping event sets. We can combine these two restrictions by requiring that A and B not be entangled

Definition 3.7. $A \rightarrow B \Leftrightarrow \exists x \in A, y \in B \mid x \rightarrow y \wedge A \nleftrightarrow B$ (New Happens-Before)

One example of a pair of event sets that satisfies the original definition of happens before but not the new definition are event sets A_1 and B_1 in Figure 3.1. (By the original definition, both $A_1 \rightarrow B_1$ and $B_1 \rightarrow A_1$.) A_1 and B_1 do not satisfy the new definition as the new definition excludes crossing or overlapping event sets.

Before continuing, we define the concurrent operator (\parallel) for event sets formally.

Definition 3.8. $A \parallel B \Leftrightarrow \forall x \in A, y \in B, x \parallel y$ (Concurrent)

Given the changes to the framework defined above, we can develop the following theorem that allows us to fully classify all possible pairs of event sets

Theorem 3.9. *Given any two event sets, A and B , their relationship can be described by exactly one of these four relationships: $A \rightarrow B$, $B \rightarrow A$, $A \parallel B$, or $A \leftrightarrow B$.*

Proof. First we will show that no two event sets, A and B , can simultaneously satisfy more than one of the four relationships above. That is, we will show that any pair of event sets satisfies at most one of the four relationships above.

Case 1.1: Assume $A \rightarrow B$. By definition, we have that $A \nleftrightarrow B$, thus $B \nrightarrow A$ since we assumed $A \rightarrow B$. Also, we know that $A \nparallel B$ since our assumption that $A \rightarrow B$ means that $\exists x \in A, y \in B \mid x \rightarrow y$. Finally we know that $A \nleftrightarrow B$, since it is part of the definition of $A \rightarrow B$.

Case 1.2: Assume $B \rightarrow A$. Since the order of operands in $A \parallel B$ and $A \leftrightarrow B$ does not matter, we can use similar reasoning as above to show that $A \nparallel B$ and $A \nleftrightarrow B$.

Case 1.3: Assume $A \parallel B$. From the definition of *concurrent*, we know that A is disjoint from B . We also know that no event in A precedes any event in B . Using only these two facts, we have that $A \nleftrightarrow B$. Thus, we have shown that no two event sets, A and B , can satisfy more than one of the four relationships.

Next, we must show that every pair of event sets, A and B satisfies at least one of the four relationships above. We know from partial-order theory that, for any two *primitive* events a and b , either $a \rightarrow b$, $b \rightarrow a$, $a \parallel b$, or $a = b$.

Case 2.1: Assume we have events sets A and B where $\exists x \in A, y \in B \mid x = y$. Clearly, we have that $A \leftrightarrow B$ regardless of what other relationships are present

among the other primitive events in A and B . For the remaining cases, we will assume that A is disjoint from B .

Case 2.2: Assume we have event sets A and B where $\exists x \in A, y \in B \mid x \rightarrow y$. There are two subcases: we additionally have that $\exists x_1 \in A, y_1 \in B \mid y_1 \rightarrow x_1$ (first subcase) or $\forall x_2 \in A, y_2 \in B, y_2 \nrightarrow x_2$ (second subcase). (Again, we have no other restrictions on the relationships among the primitive events of A and B .) In the first subcase, we have that $A \leftrightarrow B$ since we have met the requirements for A *crosses* B . In the second subcase, we have that $A \rightarrow B$ since our initial assumption that $\exists x \in A, y \in B \mid x \rightarrow y$ coupled with the assumptions that A does not cross B and A is disjoint from B satisfies the new definition of *happens before*. Using the same arguments, but starting with the assumption that we have event sets A and B where $\exists x_3 \in A, y_3 \in B \mid y_3 \rightarrow x_3$, the two subcases will yield that either $A \leftrightarrow B$ or $B \rightarrow A$.

Case 2.3: If none of the above assumptions are true, then we must have that $\forall x \in A, y \in B, y \nrightarrow x, x \nrightarrow y, x \neq y$. This is exactly the event-set definition of concurrent, and thus in this case, $A \parallel B$.

We have proven that exactly one of the four above relationships holds for every possible pair of event sets, A and B . □

The new framework as described above has remedied two of the problems with the original event framework. Contradictions where $A \rightarrow B$ and $B \rightarrow A$ can no longer occur in the new framework as proven by Theorem 3.9. We can also now recognize overlapping and crossing event sets and have shown by Theorem 3.9 that recognizing these additional types of event relationships means that now any pair of event sets has a well-defined, unique relationship.

3.2 Efficient Detection of Event Entanglement (and more)

Entanglement is precisely defined in the previous section in Definition 3.5. If we use this definition in the most obvious way to check whether two event sets, A and B , are entangled, we may end up examining $|A| * |B|$ pairs of events. Checking large event sets in this way could be time-consuming and so we desire a more efficient way to determine whether two event sets are entangled. Before we continue, we present a few relevant definitions.

Definition 3.10. Let $\text{loc}(A)$ represent the complete set of traces represented by the events in A (called the *location set* of A).

Definition 3.11. The greatest predecessors of an event, a , denoted $\text{GP}(a)$ are $\bigcup_p \text{GP}(a, p)$ (see also Definition 2.1).

Definition 3.12. The greatest predecessors of an event set, A , denoted $\text{GP}(A)$ are formed by computing $\bigcup_p \bigcup_{a \in A} \text{GP}(a, p)$ and only keeping the most-recent event per trace.

Definition 3.13. Let $\lfloor A \rfloor_p$ represent the set containing the least-recent event on trace p in event set A .

Definition 3.14. Let $\lceil A \rceil_p$ represent the set containing the most-recent event on trace p in event set A .

Definition 3.15. Let $\lfloor A \rfloor = \bigcup_{p \in \text{loc}(A)} \lfloor A \rfloor_p$ (called the *location-set front* of A)

Definition 3.16. Let $\lceil A \rceil = \bigcup_{p \in \text{loc}(A)} \lceil A \rceil_p$ (called the *location-set back* of A)

Checking if two event sets are entangled involves verifying that the event sets either overlap or cross. Since previous work explicitly assumed that two event sets were disjoint [7, 58] and implicitly assumed that event sets did not cross, efficient

ways of checking for entangled event sets were not developed. Methods do exist for non-entangled event sets that determine the precedence relationship between both *convex* event sets and arbitrary event sets.

For convex event sets, it is possible to determine the relationship between the non-entangled event sets using special timestamps assigned by the following algorithm.

Algorithm 3.17. *The timestamp T_E for a convex event E is calculated as follows. The timestamps, T_e , of the primitive events, e , of E are used and p_e is the trace on which event e is located.*

1. *Assign the element-wise maximum over all $T_e : e \in E$ to T_E*
2. *For each position in T_E corresponding to a trace, s , in the location set, set the value of that position (i.e., $T_E[s]$) equal to $T_e[s]$ where e is the earliest event in E on trace s .*

Intuitively, the resulting timestamp indicates, for each of the positions corresponding to processes in the location set (3.10), the index value of each of the events in the location-set front (3.15), and the remaining timestamp positions indicate the greatest predecessors (3.12) of the events in the location-set back (3.16). Once we have timestamped the two event sets, we can compare the two timestamps to determine the precedence relationship that exists between the two sets. If we can find a pair of elements from the two event sets' timestamps at a particular index position that are non-zero and not equal, then the event set with the smaller of these two values happened before the other event set. Otherwise, the event sets are concurrent. Unfortunately, we can show through a trivial example that the information in this type of timestamp alone is insufficient for determining event-set entanglement. Figure 3.3 shows two event sets that are not entangled; whereas,

Figure 3.4 shows two entangled event sets. In each figure, the timestamp for event set A is the same as positions corresponding to processes in the location set are represented only by the least-recent event per process. As a result, this type of timestamp can only determine precedence properly between two event sets that are known to be not entangled.

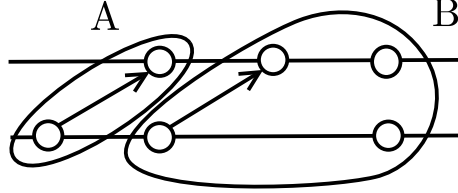


Figure 3.3: Non-entangled Event Sets

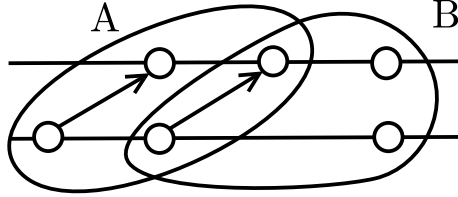


Figure 3.4: Entangled Event Sets

Alternatively, for determining precedence between arbitrary (non-convex) event sets, Basten et al [7] proposed comparing the timestamp of each event in the location-set front (3.15) of one event set against the element-wise maximum of the timestamps of the other event set (and vice versa). We will use a more efficient variation on this approach to determine event-set entanglement.

Theorem 3.18. $A \leftrightarrow B \Leftrightarrow \exists p_0, p_1, q_0, q_1 \mid ([A]_{p_0} \rightarrow [B]_{q_0} \wedge [B]_{q_1} \rightarrow [A]_{p_1}) \vee [A]_{p_0} = [B]_{p_0} \vee [B]_{q_1} = [A]_{q_1}$.

Proof. The proof relies heavily on the definition of entanglement (3.5). For conciseness, define $A \dashrightarrow B$ to mean that either $A \rightarrow B$ or $A = B$. First, assume $A \leftrightarrow B$.

By Definition 3.5, we know that A *crosses* B or A *overlaps* B .

Case 1.1: Assume A *crosses* B but A is disjoint from B . From Definition 3.4, we know that $\exists x_0 \in A, y_0 \in B \mid x_0 \rightarrow y_0 \wedge \exists x_1 \in A, y_1 \in B \mid y_1 \rightarrow x_1$. Let p_0, p_1, q_0, q_1 be the traces containing x_0, x_1, y_0, y_1 , respectively. We know that $\lfloor A \rfloor_{p_0} \dashrightarrow x_0$, $y_0 \dashrightarrow \lceil B \rceil_{q_0}$, $\lfloor B \rceil_{q_1} \dashrightarrow y_1$, and $x_1 \dashrightarrow \lceil A \rceil_{p_1}$. Since transitivity of the happens-before operator holds for individual events (but not always for event sets), we can combine all the happens-before relationships and have that $\lfloor A \rfloor_{p_0} \rightarrow \lceil B \rceil_{q_0}$ and $\lfloor B \rceil_{q_1} \rightarrow \lceil A \rceil_{p_1}$, which is what we require.

Case 1.2: Assume A *overlaps* B . Let event x on trace p be common to A and B . If $x = \lfloor A \rfloor_p = \lceil B \rceil_p$ or $x = \lfloor B \rfloor_p = \lceil A \rceil_p$ then we have what we require. Otherwise, we must have that either $\lfloor A \rfloor_p \rightarrow x$ and $x = \lceil B \rceil_p$, $\lfloor A \rfloor_p = x$ and $x \rightarrow \lceil B \rceil_p$, or $\lfloor A \rfloor_p \rightarrow x$ and $x \rightarrow \lceil B \rceil_p$. Again, by transitivity, we have that $\lfloor A \rfloor_p \rightarrow \lceil B \rceil_p$. Similarly, we also have that $\lfloor B \rfloor_p \rightarrow \lceil A \rceil_p$, which, together, is what we require.

Now, assume that $\exists p_0, p_1, q_0, q_1 \mid (\lfloor A \rfloor_{p_0} \rightarrow \lceil B \rceil_{q_0} \wedge \lfloor B \rfloor_{q_1} \rightarrow \lceil A \rceil_{p_1}) \vee \lfloor A \rfloor_{p_0} = \lceil B \rceil_{p_0} \vee \lfloor B \rfloor_{q_1} = \lceil A \rceil_{q_1}$.

Case 2.1: Assume $\exists p_0, p_1, q_0, q_1 \mid \lfloor A \rfloor_{p_0} = \lceil B \rceil_{p_0} \vee \lfloor B \rfloor_{q_1} = \lceil A \rceil_{q_1}$. In this case, we have overlap between A and B , and so $A \leftrightarrow B$.

Case 2.2: Assume $\exists p_0, p_1, q_0, q_1 \mid \lfloor A \rfloor_{p_0} \rightarrow \lceil B \rceil_{q_0} \wedge \lfloor B \rfloor_{q_1} \rightarrow \lceil A \rceil_{p_1}$. In this case, we have an event from A that happens before an event from B and also an event from B that happens before an event from A . Thus, the events from A and B either cross or overlap, and so $A \leftrightarrow B$. \square

Theorem 3.19. *If $\forall p, q, \lfloor A \rfloor_p \nrightarrow \lceil B \rceil_q$ then $\forall x \in A, y \in B, x \nrightarrow y$*

Proof. We will prove the above theorem by contradiction. Assume that $\exists x_0 \in A, y_0 \in B, x_0 \rightarrow y_0$. We know there is an event $x_1 \in \lfloor A \rfloor \mid x_1 = x_0$ or $x_1 \rightarrow x_0$. Likewise, we know there is an event $y_1 \in \lceil B \rceil \mid y_0 = y_1$ or $y_0 \rightarrow y_1$. Combining these facts gives us that $x_1 \rightarrow y_1$, which contradicts the left side of the theorem. \square

If we look closely at the checks done when evaluating the right side of Theorem 3.18, we might notice that, even if we determine that two event sets are not entangled, the results of some of the checks may help us identify other relationships between the two event sets. (Theorem 3.9 and Theorem 3.19 help us here.) For example, if we discover that event sets A and B are not entangled, but $\exists p_0, q_0 \mid \lfloor A \rfloor_{p_0} \rightarrow \lceil B \rceil_{q_0}$, then we know that $A \rightarrow B$. Likewise, if we discover that A and B are not entangled, but $\exists p_1, q_1 \mid \lfloor B \rfloor_{q_1} \rightarrow \lceil A \rceil_{p_1}$, then we know that $B \rightarrow A$. Finally, if none of the individual checks done when evaluating the right side of Theorem 3.18 are true, then we know that $A \parallel B$. We have thus informally proven the following Theorem.

Theorem 3.20. *Given two event sets A and B we can determine the relationship between them by evaluating at most the $2 * \text{loc}(A) * \text{loc}(B)$ event pairs formed by taking the cross product of the location-set front of A and location-set back of B and also the location-set back of A and location-set front of B and checking these pairs both for a happens-before relationship and for equality. Either*

1. *One of the three disjunctions of the right side of Theorem 3.18 will hold (meaning event sets are entangled), or*
2. *Exactly one of the two conjunctions of the first disjunction will hold, and neither of the second or third disjunctions (meaning $A \rightarrow B$ or $B \rightarrow A$), or*
3. *Neither of the conjunctions nor the second or third disjunctions will hold (meaning $A \parallel B$).*

Thus, we have demonstrated an efficient way not only to test for entanglement of two event sets, but also to completely determine any relationship between the event sets. It is interesting to note that if $A \parallel B$ then we will need to check all $2 * \text{loc}(A) * \text{loc}(B)$ event pairs. In contrast, if $A \leftrightarrow B$ we may need to check only one event pair (if A and B overlap).

3.3 Convex Closure and Its Validity

A convex event set, defined formally in Chapter 2, has a variety of uses. Convex event sets can be assigned timestamps using Algorithm 3.17. These timestamps allow us to determine precedence quickly between two such event sets (assuming no entanglement). Convex event sets can facilitate visualizing sets of events on-screen and can also be used in pattern search.

Definition 3.21. An *intervening event* of an event set, A , is an event c , such that $\exists a, b \in A \mid a \rightarrow c \wedge c \rightarrow b$.

The goal of a convex-closure algorithm is to take a set of events and add the minimum number of additional events to this set such that the defined requirements of a convex event set are satisfied. In other words, it adds to the original set any intervening events of the set. Efficient ways of implementing such an algorithm are explored in detail in Chapter 4. Expressing the convex-closure algorithm in this manner, we can state this theorem without additional proof.

Theorem 3.22. *There is a unique convex closure for any event set.*

Figure 3.5 shows a set of input events as filled circles. Figure 3.6 shows the result of a convex-closure operation performed on this input event set. The intervening events are shown as thick circles and an outline is drawn around the convex set.

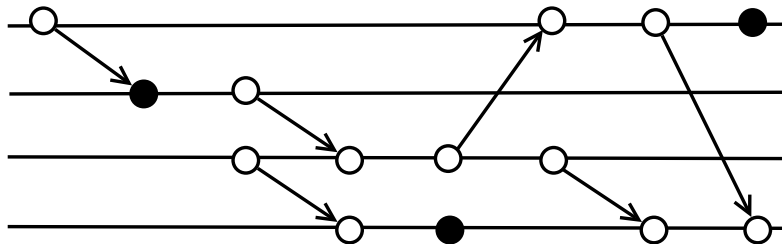


Figure 3.5: Original Event Set

$y_1 \in B$ happens before an event in $A_{original}$ and this contradicts our earlier assumption that $A_{original} \leftrightarrow B$. Using a similar style of argument we instead assume, again for the purpose of contradiction, that A_{convex} does overlap with B . Assume x_3 is the event common to both A_{convex} and B . Either $x_3 \in A_{original}$ (meaning $A_{original}$ overlaps with B giving a contradiction) or $\exists x_2 \in A_{original} \mid x_3 \rightarrow x_2$ (meaning $A_{original}$ crosses B , again giving a contradiction). Thus we also must have that $A_{convex} \leftrightarrow B$.

Implication 2: We can use the same argument as in Implication 1, but reverse the direction of each happens-before operator.

Implication 3: Assume $A_{original} \parallel B$ and show that $A_{convex} \parallel B$. Assume, for the purpose of contradiction, that $A_{convex} \not\parallel B$. This means that $\exists x_0 \in A_{convex}, y_0 \in B \mid x_0 \rightarrow y_0$, $\exists x_1 \in A_{convex}, y_1 \in B \mid y_1 \rightarrow x_1$, or $\exists x_2 \in A_{convex}, y_2 \in B \mid x_2 = y_2$ (or any combination of these). Looking at the first of these three possibilities, we have that either $x_0 \in A_{original}$ or $\exists x_3 \in A_{original} \mid x_3 \rightarrow x_0$, meaning something in A_{convex} happens before something in B (a contradiction). A similar argument applies for the second of these three possibilities. For the final possibility, we again have that either $x_2 \in A_{original}$ or $\exists x_4 \in A_{original} \mid x_4 \rightarrow x_2$. Again, this means that either $A_{original}$ overlaps with B or an event in $A_{original}$ happens before an event in B (a contradiction in either case).

Implication 4: Assume $A_{original} \leftrightarrow B$ and show that $A_{convex} \leftrightarrow B$. If $A_{original}$ overlaps B , then A_{convex} overlaps B since $A_{original} \subseteq A_{convex}$. If $A_{original}$ crosses B , we have that $\exists x_0 \in A_{original}, y_0 \in B \mid x_0 \rightarrow y_0 \wedge \exists x_1 \in A, y_1 \in B \mid y_1 \rightarrow x_1$. Again, since $A_{original} \subseteq A_{convex}$, we must have that either A_{convex} crosses B or A_{convex} overlaps B (if any of the additional events in A_{convex} are also in B).

We have only shown the proof of each implication in only one direction; however, using earlier results, it is not too difficult to show each implication holds in the

other direction as well. First, assume we have any convex set A_{convex} and any one of its corresponding original sets, $A_{original}$. Now, further assume that we have any other event set B and that $A_{convex} ? B$ where $?$ is one of the four possible event-set relationships as described in Theorem 3.9. Also assume that $A_{original} ?' B$ where $'$ is any one of the *other* three possible event-set relationships. We have already proved that $A_{original} ?' B \Rightarrow A_{convex} ?' B$, thus we have that both $A_{convex} ? B$ and $A_{convex} ?' B$. By Theorem 3.9, we have a contradiction, since $?$ and $'$ are different event-set relationships, and we know that there exists a unique event-set relationship between any two sets of events. No matter which convex event, or which one of its corresponding original event sets, or which event-set relationships we choose for $?$ and $'$, where $?$ \neq $'$, we will always arrive at a contradiction. Thus, we must have that $?$ $=$ $'$, again by Theorem 3.9, since every pair of event sets must satisfy exactly one of the four possible relationships. \square

Although a convex event, A_{convex} , obtained by applying a convex-closure algorithm to event set $A_{original}$ is a faithful substitute for $A_{original}$ with respect to the standard event-set relationships, it should be pointed out that this does not always hold for the sub-relationships of the entanglement operator (cross and overlap). In some cases, $A_{original}$ will cross an event set B , but A_{convex} (created by taking the convex closure of $A_{original}$) will overlap (rather than cross) the same event set B . Although, clearly, if $A_{original}$ overlaps B then A_{convex} will as well. If, in certain future scenarios, it becomes necessary to make a distinction between crossing and overlapping event sets, then it would be prudent to check whether applying the convex-closure algorithm to event sets will distort the intended semantics.

3.4 Additions to Pattern Language

The first part of this chapter focused on presenting results related to evaluating relationships between two sets of events. The remainder of the chapter builds on these results, but broadens our focus. Our broader interest is determining which events satisfy a given pattern. In this chapter’s context of providing a framework, we will focus on describing which features of a pattern language are desirable; that is, which give us the ability to specify a wide variety of useful patterns. The rest of this section will focus on language features that we have incorporated into the pattern language; the next section will provide useful patterns that make use of these new features.

Several pattern languages based on partial-order operators have been defined [13, 29, 33, 36, 51]. Many of the useful features of these languages were incorporated into a new language specified by Xie [58, 64]; however, other useful features that can add significant expressiveness to the language are missing. Below, we describe the new language features and provide a high-level description of how patterns are evaluated. We also provide a complete grammar for the new language.

The first new feature we add to the existing pattern language is variable binding. In the original pattern language, we can only specify classes of variables and the relationships that need to be satisfied among events that satisfy the constraints of those classes. For example, evaluating the simple pattern $\mathcal{A} \rightarrow \mathcal{B}$ will return all pairs a, b where a satisfies the constraints of event class \mathcal{A} , b satisfies event class \mathcal{B} and $a \rightarrow b$. If we instead want to return all sets a, b, c where a , b , and c satisfy, respectively, the constraints of classes \mathcal{A} , \mathcal{B} , and \mathcal{C} and where not only $a \rightarrow b$, but $a \rightarrow c$ then we need variable binding. In the original pattern language, we can attempt to compose a pattern $\mathcal{A} \rightarrow \mathcal{B} \wedge \mathcal{A} \rightarrow \mathcal{C}$; however, the processing of this pattern will consider each of the occurrences of \mathcal{A} separately and will return

two events, a_1 and a_2 , one that satisfies the first constraint (that it happens before something in class \mathcal{B}) and one that satisfies the second constraint (that it happens before something in class \mathcal{C}). Using variable binding, we are able to declare a variable a of type \mathcal{A} and then the pattern $a \rightarrow \mathcal{B} \wedge a \rightarrow \mathcal{C}$ will give the desired result. Without variable binding, it is simply not possible to search for patterns like this.

In the original pattern language, there is no way to specify exactly what should be returned when a successful match has been discovered. Thus, once a match is found, an event set is returned containing *every* event that was matched to a class or variable in the pattern. Unfortunately, this is not always the desired behaviour. In some cases, we are interested only in the events that match a certain subset of the classes or variables—relationships involving the other classes or variables simply serve as additional restrictions on the ones of interest. Going back to our previous example, $a \rightarrow \mathcal{B} \wedge a \rightarrow \mathcal{C}$, our goal may simply be to discover all events of type \mathcal{A} that happen before both an event of class \mathcal{B} and an event of class \mathcal{C} . We may not care which specific events in class \mathcal{B} or \mathcal{C} are involved in the match. A feature that allows us to specify exactly which events are returned after each successful match would be beneficial.

Finally, we may sometimes want to express a pattern using universal quantifiers. We could, for example, want to find each event of type \mathcal{A} that happens before *all* events of type \mathcal{B} . Figure 3.7 shows a set of events that are either of type \mathcal{A} (filled) or \mathcal{B} (outlined). The circled events are the events of type \mathcal{A} that would be returned in this case. Because the original pattern language does not provide for this, we add a feature that allows us to specify that a universal quantifier should be applied to a given variable.

Figure 3.8 shows a grammar for the new pattern language. It is based on Xie’s grammar, but incorporates the new features outlined above. It also adds support


```

predicates ⇒ (predicate “;”) *
predicate ⇒ id “:=” clause
                | id variable (“,” variable) *
clause ⇒ term basicOperator term
                ⇒ term “!” basicOperator term
                | term booleanOperator term
basicOperator ⇒ “→”
                | “||”
                | “↔”
booleanOperator ⇒ “^”
                | “v”
term ⇒ id
                | variable
                | class
                | class.class
                | “(” clause “)”
class ⇒ “[” process “,” type “,” text “]”
                | “[” (id “=” string)
                    (“,” id “=” string) * “[”
variable ⇒ [“$”, “*”, “~”] id
id ⇒ alpha (alnum) *
alpha ⇒ [“a” – “z”, “A” – “Z”, “_”]
alnum ⇒ [“a” – “z”, “A” – “Z”, “_”, “0” – “9”]
string ⇒ [“a” – “z”, “A” – “Z”, “_”, “0” – “9”, “:”, “ ”,
                “\t”, “*”, “.”, “/”, “(”, “)” ] +

```

Figure 3.8: Grammar for the New Pattern Language

operators further restrict which events match our pattern. The “not” (!) operator can precede any of these three operators and has the effect one might expect given Theorem 3.9. For example, after defining classes \mathcal{A} and \mathcal{B} , we could write the patterns like $\mathcal{A} \rightarrow \mathcal{B}$ or $\mathcal{A}! \leftrightarrow \mathcal{B}$ (however, in this document we will always write negated operators using more aesthetic typography such as $\mathcal{A} \nleftrightarrow \mathcal{B}$).

To build even more complex patterns, it is possible to combine multiple independent constraints (as described in the previous paragraph) with Boolean operators \wedge and \vee , and make the constraints themselves more complex by nesting multiple levels of operators. We can now write, if we define variables $\$a$ for class \mathcal{A} and $\$b$ for class \mathcal{B} , patterns such as $((\$a \rightarrow \$b) \wedge (\$a \rightarrow \mathcal{C})) \vee (\$b \rightarrow \mathcal{C})$ or $((\$a \rightarrow \mathcal{B}) \parallel \mathcal{C}) \wedge (\$a \rightarrow \mathcal{C})$.

We now discuss how a pattern like $((\$a \rightarrow \mathcal{B}) \parallel \mathcal{C}) \wedge (\$a \rightarrow \mathcal{C})$ is evaluated. Without considering any optimizations, we can start by simply enumerating all possible assignments of events to the variables and class identifiers. For example, our event set may contain four events (events 1, 2, 3, and 4) that are part of class \mathcal{A} . Each of these four events is a potential match for the variable $\$a$ in our pattern. A match to the *entire* pattern will involve a set of events where one event matches $\$a$, one event matches \mathcal{B} , one matches the first \mathcal{C} and one matches the second \mathcal{C} . (Note that $\$a$ represents the same event wherever it appears; whereas \mathcal{C} potentially represents a different event each time it appears.) The maximal set of matches for any pattern is equivalent to the cross product of the sets of events that can be matched to each variable or class identifier. Most of the time the set of matches returned will be much smaller than the maximal set; however, the maximal set provides us a way to evaluate the pattern. To do so, we can assign one potential match from the maximal set to the variables or class identifiers in a pattern. If the potential match satisfies the pattern, then we return it; otherwise we do not. The next potential match from the maximal set is then examined. Boolean operators in a pattern are treated similarly to the way they are treated when appearing in a

conditional statement in most modern programming language.

The new features added to the pattern language complicate this approach. For example, a variable may be marked with a tilde (\sim), indicating that the event assigned to it should not be returned as part of the match. Adapting our approach to include this feature is simple. We simply check after discovering a successful match which elements should or should not be returned. For example, evaluating the pattern $\mathcal{A} \rightarrow \sim b$ would start by enumerating the maximal set of matches (“all events in class \mathcal{A} ” cross “all events in class \mathcal{B} ”). It would then discover which pairs of events from this maximal set satisfy the pattern and, for those pairs that did, return only the event from the pair that matches to \mathcal{A} . The other event in the pair, which matches to $\sim b$, is discarded. The effect of this pattern is to return any events in class \mathcal{A} that happen before any event in class \mathcal{B} . Recall that we are not considering the efficiency of the approach at this point.

Another new feature, the universal quantifier, poses more of a challenge. When a universal quantifier, indicated with an asterisk (*), is applied to individual variables, it means that the event assigned to the other variables or class identifiers in the pattern (those not marked with the universal quantifier) must satisfy the pattern for all possible values of the variables marked with the asterisk. When creating our maximal set of matches, we should only include those variables not marked with this quantifier. We then pick one potential match from our maximal set and assign the events to the appropriate variables or class identifiers. Now, we must evaluate the pattern multiple times, each time picking a new value for one of the variables marked with an existential qualifier. If the entire pattern is satisfied *every* time, then we can output the potential match we initially chose from our maximal set. It is important to note that we only require that the *entire pattern* be satisfied for a given universally-quantified variable. (It does not have to be satisfied for each individual sub-expression.) For example, the semantics of $\mathcal{B} \rightarrow *a \vee \mathcal{C} \rightarrow *a$ are that

every event set matching the clause we declared for $*a$ needs to be preceded by *either* the chosen event from class \mathcal{B} or the event from class \mathcal{C} , since the two subexpressions are in a disjunction. Conversely, $\mathcal{B} \rightarrow *a \wedge \mathcal{C} \rightarrow *a$ would require every event set matching the clause we declared for $*a$ to satisfy both of the subexpressions, for the chosen events from classes \mathcal{B} and \mathcal{C} , since the subexpressions are in a conjunction.

It is possible with these new features that no events will be output after a successful match. For example, the pattern $\sim a \rightarrow \sim b$ simply checks that there is at least one event from class \mathcal{A} that happens before at least one event from class \mathcal{B} (assuming we have declared $\sim a$ as class \mathcal{A} and $\sim b$ as class \mathcal{B}). The pattern $*a \rightarrow *b$ checks that every event from class \mathcal{A} happens before every event from class \mathcal{B} (assuming we have declared $*a$ as class \mathcal{A} and $*b$ as class \mathcal{B}). Since returning nothing is ambiguous, the pattern-search algorithm should return either that the pattern was successfully matched or that it was not.

It should also be noted that $*$ and \sim can be applied only to variables (as indicated in the grammar). These modifiers are not permitted directly on clauses for two reasons. First, patterns are more readable if the modifiers are applied to variables as the scope of the modifier is immediately visible. Second, we avoid confusing patterns that are not well defined like $*(\$a \rightarrow \$b) \wedge (\$a \rightarrow \$c)$. In evaluating such a pattern, the value of the second occurrence of $\$a$ is unclear. Since variables are limited in scope to a single predicate (as defined in the grammar), expressing this pattern as $*x \wedge y$ where x is a variable declared as the class representing predicate $(\$a \rightarrow \$b)$ and y is a variable declared as the class representing predicate $(\$a \rightarrow \$c)$, will avoid the problem above as the $\$a$ in each of x and y do not refer to the same variable.

Finally, one feature that is not present in this pattern language, but was present in earlier languages on which this language is based, is the limited operator. The expression $\mathcal{A} \xrightarrow{\mathcal{C}} \mathcal{B}$, which uses the limited operator, returns any matches to \mathcal{A}

and \mathcal{B} where no occurrence of an event matching \mathcal{C} happens both after the match to \mathcal{A} and before the match to \mathcal{B} . Since our new language offers variable binding and allows universal quantifiers, we can instead specify this expression as $\mathcal{A} \rightarrow \mathcal{B} \wedge (\mathcal{A} \nrightarrow *c \vee *c \nrightarrow \mathcal{B})$. Although the limited operator is not officially mentioned in the grammar, we can still choose to recognize it and immediately convert it to the equivalent expression using more general constructs.

In Chapters 5 and 6 we will continue our discussion about how to evaluate patterns with a focus on practical and efficient ways to do so.

3.5 Completeness of Operator Set

Ideally, we would like to present a formal comparison of the expressiveness of the pattern language before and after adding the additional features from the previous section. Unfortunately, formally analyzing the expressiveness of any language is a very complex pursuit and is beyond the scope of this thesis. The primary rationale for adding the additional pattern-language features is to allow us to present more elaborate and relevant examples by which we can demonstrate performance differences among the various algorithms presented in subsequent chapters.

Although a formal comparison is not feasible, this final section of Chapter 3 informally demonstrates the usefulness of some of the additional features that were added to the pattern language. We provide a few practical patterns expressible with the new language features with the goal of providing a convincing, although not formal, argument of how the additional features noticeably improve the expressiveness of the pattern language.

Note that these patterns are expressed exactly as they would be in a text file provided to the search algorithm. As such, some of the operators need to be expressed in a text-friendly way. \wedge and \vee operators are represented as $\&$ and $|$, \rightarrow ,

\parallel and \leftrightarrow are represented as $->$, \parallel , and $<->$, and preceding an operator with “!” indicates it should be negated (i.e., \nrightarrow is written as $!->$).

All the patterns will focus on data collected from a TCP-socket target environment, but many are general enough to be adapted for use in other environments. The scenario from which the data is collected involves two client processes connecting to a server process. Each client then sends a large number of pieces of data to the server and finally closes the connection. Each process and each socket is represented by a trace. A client can connect to the listener socket of the server through a series of events as illustrated in the circle-shaped events in Figure 3.9. Once connected, a server can transfer data by sending it to the socket trace. The data is then moved from the socket trace to the client trace and such data transfers are shown by the outlined-square-shaped events. Note that for simplicity, only one client is shown in the figure, yet our examples will assume two clients are involved.

Table 3.1 shows a portion of the internal data POET collected from a TCP-sockets application. Each event has only a small number of attributes. The type is a coded value, e.g., type 3 is “create socket”. Two trace and index values are included, for the event itself and possibly for the partner event, where a -1 trace value indicates the partner data is missing. Text effectively provides an annotation for the event. Note that the real time at which an event occurred is also recorded, but is not shown in the table. Although this data representation allows only a limited number of attributes, work has been done recently to allow for a richer representation of data [50].

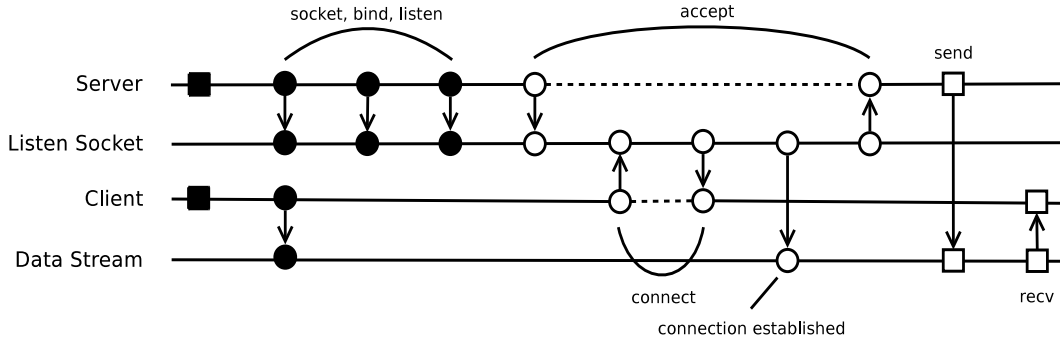


Figure 3.9: TCP-Socket Environment

Type	Trace	Index	Partner		Text
			Trace	Index	
1	0	0	-1	0	Host: orinoco PID: 9751
3	0	1	-1	0	Created socket: Host: orinoco PID: 9751 FD: 16
4	1	0	0	1	New socket: IP: 9.26.109.213 PID: 9751 FD: 16
1	2	0	-1	0	Host: orinoco PID: 9770
3	2	1	-1	0	Created socket: Host: orinoco PID: 9770 FD: 18
4	3	0	2	1	New socket: IP: 9.26.109.213 PID: 9770 FD: 18
1	5	0	-1	0	Host: orinoco PID: 9776
3	5	1	-1	0	Created socket: Host: orinoco PID: 9776 FD: 20
4	6	0	5	1	New socket: IP: 9.26.109.213 PID: 9776 FD: 20
6	0	2	-1	0	Bound to port: 8123
7	1	1	0	2	Bound to IP: 9.26.109.213 Port: 8123
9	0	3	-1	0	Listening on port 8123
10	1	2	0	3	Listening on IP: 9.26.109.213 Port: 8123
15	0	4	-1	0	Accepting on: IP: 9.26.109.213 Port: 8123

Table 3.1: Internal Representation of Events (TCP-Socket Environment)

The goal of our first set of patterns (see Figure 3.10) is to discover the first and last established connections. We can find these occurrence using the modified pattern language by using variable binding, universal quantifiers (asterisk), and the not operator. (None of these features were available in the previous partial-order-based pattern language, although part of this particular pattern could have been written using the old limited operator described in Section 3.4.) First, we must

define the events that indicate a connection has been established. The connection-establishment sequence is made up of five pairs of events. Rather than searching for all of these pairs, we can instead search for the first (`$sc`) and last (`$dc`) pair, with the extra constraint that a second occurrence of the first pair (`*sc_all`) cannot occur between the first and last pair. Specifically, we require that either `$sc` does not happen before `*sc_all` or that `*sc_all` does not happen before `$dc`. Recall that variables marked with the universal quantifier (as well as existentially-qualified variables) only need to be satisfied over the *entire pattern* and not necessarily in each subpattern in which they appear.

Now that we have a pattern to find the establishment of a connection, we need to write a pattern that will determine which of these connection-establishment sequences happened first or last using the `FirstConnectionEstablished` and `LastConnectionEstablished` patterns. In a partial-order context, we know that if no connection-establishment (CE) event set happens before a specific CE event set, then that specific one happened first and that if no CE event sets happen after a specific CE event set, then that specific one happened last. Note that there may be multiple concurrent sets of events that are the “first” or “last” connection established.

For our second set of patterns, shown in Figure 3.11, we show how to discover the last data transfer to the server across both of our clients. Again, we specify a data-transfer class for each of the two clients (`DataTransferC1` and `DataTransferC2`). We then use a similar approach as in the “`LastConnectionEstablished`” pattern from the previous example. We can first search for `FinalDataTransferC1`, which will return a match if client 1 performed the final data transfer. If not, then we can search for `FinalDataTransferC2`, which will return a match if client 2 performed the final data transfer.

Our third set of patterns, shown in Figure 3.12, attempts to find all occurrences

```

StartConnect := ["Server", "Accept", ""].
               ["Listen Socket", "Accept_stream", ""];
DoneConnect := ["Server", "Accept_done", ""].
               ["Listen Socket", "Accept_done_stream", ""];
StartConnect $sc, *sc_all;
DoneConnect $dc;

ConnectionEstablished := ($sc --> $dc) &
                        (($sc !--> *sc_all) |
                         (*sc_all !--> $dc));
ConnectionEstablished *ce_all;

FirstConnectionEstablished := *ce_all !-->
                             ConnectionEstablished;

LastConnectionEstablished := ConnectionEstablished !-->
                             *ce_all;

```

Figure 3.10: Example Pattern 1

```

DataTransferC1 := ["Client1", "Send", ""].
                 ["DataStream1", "Send_stream", ""];
DataTransferC2 := ["Client2", "Send", ""].
                 ["Data Stream2", "Send_stream", ""];

DataTransferC1 $dtc1, *alldtc1;
DataTransferC2 $dtc2, *alldtc2;

FinalDataTransferC1 := ($dtc1 !--> *alldtc1) &
                      ($dtc1 !--> *alldtc2);
FinalDataTransferC2 := ($dtc2 !--> *alldtc1) &
                      ($dtc2 !--> *alldtc2);

```

Figure 3.11: Example Pattern 2

```

DataTransferC1 $dtc11, $dtc12, *dtc1_all1, *dtc1_all2;
DataTransferC2 $dtc21, $dtc22, *dtc2_all1, *dtc2_all2;

ConsecutiveC1Sends := ($dtc11 --> $dtc12) &
                      (($dtc11 !--> *dtc1_all1) |
                       (*dtc1_all1 !--> $dtc12)) &
                      (($dtc11 !--> *dtc2_all1) |
                       (*dtc2_all1 !--> $dtc12));
ConsecutiveC2Sends := ($dtc21 --> $dtc22) &
                      (($dtc21 !--> *dtc1_all2) |
                       (*dtc1_all2 !--> $dtc22)) &
                      (($dtc21 !--> *dtc2_all2) |
                       (*dtc2_all2 !--> $dtc22));

```

Figure 3.12: Example Pattern 3

of two consecutive send events from the same client. That is, we want to discover whenever a client places two consecutive chunks of data on the data-stream trace. We reuse our `DataTransferC1` and `DataTransferC2` classes from the previous example. We then need to write a pattern that discovers two consecutive event sets of either the `DataTransferC1` or `DataTransferC2` types. The `ConsecutiveC1Sends` pattern does this for client 1, and the `ConsecutiveC2Sends` pattern does this for client 2. Each of these patterns ensures that two data transfers from the same client happen after each other and that no data transfers from either client happen in between (using a similar technique as the previous examples).

Our final set of patterns attempts to ensure that every data transfer from a client to a data stream is followed by a data transfer from that data stream to the server. It essentially is able to discover some occurrences of data being lost between the client and server. The patterns are shown in Figure 3.13 and we again reuse the `DataTransferC1` class from previous examples. Because we want a “yes” or “no” answer, we can model the `HappensBeforeC1C2` pattern using a tilde operator on each client-data-transfer event set. This will cause no events to be returned,

```

DataTransferS1 := ["DataStream1", "Recv_stream", ""].
                ["Server", "Recv", ""];
DataTransferS2 := ["DataStream2", "Recv_stream", ""].
                ["Server", "Recv", ""];

DataTransferC1 ~dataTc1;
DataTransferC2 ~dataTc2;
DataTransferS1 *dataTs1;
DataTransferS2 *dataTs2;

HappensBeforeC1C2 := (~dataTc1 !--> *dataTs1) |
                    (~dataTc2 !--> *dataTs2);

```

Figure 3.13: Example Pattern 4

but the pattern-search algorithm will indicate success or failure. It is easiest, in this case, to write the pattern to search for the presence of data loss, rather than the absence of it. The `HappensBeforeC1C2` pattern checks that, for each client data transfer, every corresponding server data transfer does *not* happen after it. If every corresponding server data transfer does not happen after a certain client data transfer, then we know that not even a single transfer does happen after it. Thus, if the pattern is successfully matched, we know there is a data-loss issue.

We have shown in the four sets of example patterns above how the new pattern-language features add significant expressiveness to the language and increase the variety of patterns we can search for. Although these patterns are specific to the TCP environment, many of the concepts (finding the first or last occurrence of a particular sequence, finding two consecutive sequences, or verifying concurrent behaviour between two sets of events) can be extracted and used in other environments.

3.6 Evaluating Patterns

Although evaluating patterns will be discussed in greater detail in subsequent chapters, we will present an overview in this section of some high-level ideas. Without considering efficiency, it is easy to evaluate a pattern. As alluded to in previous sections, we can pick a combination of values for the existentially-qualified variables in the pattern and then verify that all of the operators in the pattern are satisfied for that assignment to the existentially-qualified variables over all combinations of assignments to the universally-quantified variables.

The following example shows how this approach is inefficient. Assume our pattern is $(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow \mathcal{C}$. Further assume we choose an assignment for \mathcal{A} , \mathcal{B} , and \mathcal{C} , and our verification first checks the subpattern $(\mathcal{A} \rightarrow \mathcal{B})$ and fails. Our next assignment may leave \mathcal{A} and \mathcal{B} unchanged and choose a new value for \mathcal{C} . We will continue to fail on $(\mathcal{A} \rightarrow \mathcal{B})$, at least until we have exhausted all values of \mathcal{C} and are able to choose a new value for \mathcal{A} or \mathcal{B} .

We can greatly increase efficiency by moving the assignment of variables inside the pattern. For example, rather than evaluating the pattern as $\forall A, \forall B, \forall C [(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow \mathcal{C}]$, we could instead evaluate it as $\forall C [\forall A, \forall B [(\mathcal{A} \rightarrow \mathcal{B})] \rightarrow \mathcal{C}]$. In this case, we only assign a value to \mathcal{C} once we have satisfied the subpattern $(\mathcal{A} \rightarrow \mathcal{B})$. The pattern-search algorithms in the following chapters all attempt to leverage this optimization.

Chapter 4

Convex-Closure Algorithm

CONVEX EVENTS, as mentioned previously, are useful in a variety of situations. They can be used to determine precedence between two event sets (assuming no entanglement), can facilitate visualization of sets of events on-screen, and are sometimes useful in pattern search. We can also represent them more efficiently than arbitrary event sets. This chapter will describe efficient ways of calculating a convex closure, given a set of partially ordered events.

We start by examining an existing algorithm [58] for computing a convex closure and analyze its execution time. We follow this with a detailed description of a new algorithm and compare its execution to that of the existing one. Additional improvements to this new algorithm are also presented. We then briefly explore efficient ways of combining existing closures. Finally, we present empirical data that shows the benefits of the improvements.

4.1 Convex-Closure Theorems and Definitions

Before we describe the algorithms, we present a theorem and definitions that are used in one or both of the convex-closure algorithms.

Theorem 4.1. [58] *Let T_e be the existing Fidge/Mattern timestamp for a primitive event e . The index of e 's greatest predecessor on trace P_i is equal to $T_e[P_i] - 1$ if $T_e[P_i]$ is greater than 0. Otherwise, e has no predecessors on trace P_i .*

It is important to note that a convex event can be represented by two vectors (which we call *front* and *back*) rather than the entire set of primitive events contained in the convex event. The *front* vector contains the least-recent event from each trace that is part of the convex event. Similarly, the *back* vector contains a corresponding most-recent event from each trace. This is a valid way to represent convex events as every event on a given trace between the least-recent and most-recent events must be in the convex event (otherwise we would violate the definition of the convex event, E , as we would have $x \rightarrow z \rightarrow y$ where $z \notin E$). The more specific goal of each convex-closure algorithm, then, is to compute these two vectors.

The computation of the *back* vector is the most straightforward in each algorithm as we can take advantage of Theorem 4.1. There is no analogous theorem to use for computing the *front* vector as Fidge/Mattern timestamps do not encode information about an event's least successors.

The algorithms also depend on the following four definitions [58].

Definition 4.2. $S_{back}(E) := \{t \in E \mid t \text{ has no successors in } E\}$, where E is a set of primitive events.

Definition 4.3. $S_{front}(E) := \{t \in E \mid t \text{ has no predecessors in } E\}$, where E is a set of primitive events.

Definition 4.4. $T_{back}(E) := \{t \in E \mid t \text{ has no successors in } E \text{ on its trace}\}$, where E is a set of primitive events.

Definition 4.5. $T_{front}(E) := \{t \in E \mid t \text{ has no predecessors in } E \text{ on its trace}\}$, where E is a set of primitive events.

4.2 Existing Algorithm

In this section, we present an existing convex-closure algorithm in detail, describing how it computes the *back* and *front* vectors that make up the convex closure. We also provide an asymptotic analysis of its runtime.

4.2.1 Description of Existing Algorithm

Intuitively, the existing algorithm [58, 64] attempts to find any relevant events that occur between an event in S_{front} (Definition 4.3) and an event in S_{back} (Definition 4.2). Computing the *back* is easy, as the timestamps of the events in S_{back} contain information about the greatest predecessors of each event. Examining each greatest predecessor of every event in S_{back} to determine if it is a successor of any event in S_{front} is sufficient to determine which of these greatest predecessors should form the *back* vector. Intuitively, the *front* vector is computed by walking forward starting with each element of S_{front} along all paths, as in Figure 4.1. We can stop examining a given path once we encounter an event on that path is not a predecessor of any event in S_{back} .

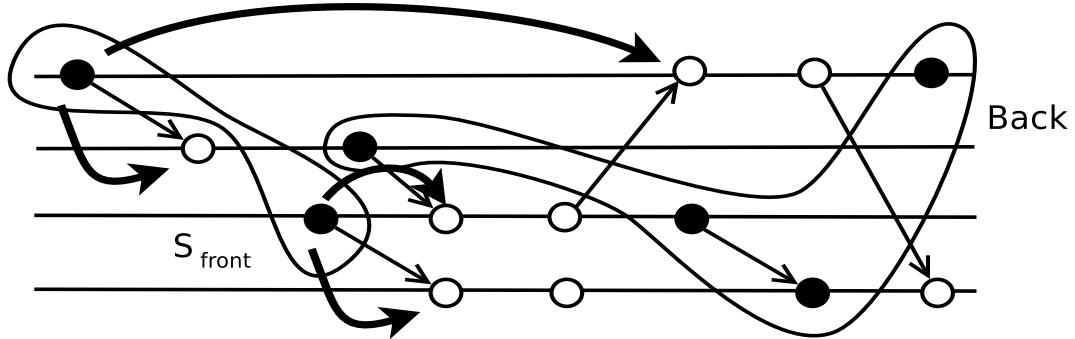


Figure 4.1: Existing Computation of the *front*

In pseudocode, the existing algorithm computes the *back* vector of E using these

steps. For this algorithm, an event's greatest predecessor on its trace is itself.

Algorithm 4.6. *Existing Computation of the Back Vector*

```

// First, find the least and most recent events in
// event set E on each trace
Sfront[1..N] = {null, ..., null}
Sback[1..N] = {null, ..., null}
for event in E
    if Sfront[event.trace] == null || event → Sfront[event.trace]
        Sfront[event.trace] = event
    if Sback[event.trace] == null || Sback[event.trace] → event
        Sback[event.trace] = event
// Eliminate any events in Sfront that happen after
// another event in Sfront
// Eliminate any events in Sback that happen before
// another event in Sback
for i = 1 to N
    for j = 1 to N
        if Sfront[i] != null && Sfront[j] != null
            && Sfront[i] → Sfront[j]
                Sfront[j] = null
        if Sback[i] != null && Sback[j] != null
            && Sback[i] → Sback[j]
                Sback[i] = null
// Having computed Sfront and Sback, we next
// examine the greatest predecessors (GP)
// of the events in Sback (see Theorem 4.1)
// We consider a GP to be valid if it happens after an event in
// Sfront but we only store a GP if we have not seen
// a more recent GP on that trace previously
back[1..N] = {null, ..., null}
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            if Sback[i] != null && Sback[i].GP[j] != null
                && Sfront[k] → Sback[i].GP[j]
                && (back[j] == null || back[j] → Sback[i].GP[j])
                    back[j] = Sback[i].GP[j]

```

The second half of the the existing algorithm computes the *front*. It uses the following steps, presented below in pseudocode. This part of the algorithm uses two

temporary variables: an event set and an event vector of size equal to the number of traces. The event set contains events that are awaiting checking and the event vector stores the least-recent event encountered thus far for each trace.

Algorithm 4.7. *Existing Computation of the Front Vector*

```

// Add all events from Sfront to eventSet and
// initialize front to null
eventSet = Sfront
front[1..N] = {null, ..., null}
// Look at one event at a time
// If the event is less recent than the one in front[event.trace]
// then replace front[event.trace] with this event
// Add any of the event's immediate successors (IS)
// IS[1] and IS[2] to the eventSet if they happen before
// an event in the back
do
    if eventSet.empty exit
    event = eventSet.remove()
    if front[event.trace] == null || event → front[event.trace]
        front[event.trace] = event
    for i = 1 .. number of IS of event
        for j = 1 .. N
            if back[j] == null continue
            if event.IS[i] → back[j]
                eventSet.add(event.IS[i])
                break

```

Previous work [58] has proven the correctness of these computations.

4.2.2 Asymptotic Behaviour of Existing Algorithm

We determine the asymptotic behaviour of the existing algorithm by examining each step in the computation of both the *back* and *front* vectors. We express the behaviour in terms of n , the number of traces in the entire data set; t , the number of traces in the input event set; e , the number of events in the input event set; and c , the number of events in the output event set (convex closure).

Step 1 of the *back*-vector algorithm requires that we examine each event in the input set and then compare every event in the least-recent set (and most-recent set) against each other. Overall, this requires $O(e + t^2)$ operations. Examining each set of greatest predecessors in steps 2 and 3 requires us to look at n greatest predecessors from each of (up to) t events in S_{back} . Furthermore, each of these nt greatest predecessors has to be compared against (up to) t events in S_{front} . Overall, this results in $O(nt^2)$ operations in the worst case (if the sizes of both $S_{back}(E)$ and $S_{front}(E)$ are t). In total, the entire computation requires $O(e + t^2 + nt^2)$ steps. Since $n \geq t$, we can also write this as $O(e + n^3)$.

In the computation of the *front* vector, every event that forms the convex closure is examined at some point (we only stop exploring a path when we discover an event that is *not* in the closure). In addition, one extra event per event in the convex closure may be examined and discarded, since an event can have at most two immediate successors, and we continue examining a given path if at least one of the two successors is not discarded. When we reach the end of a given path, we can discard up to two events; we will, however, only reach the end of a given path at most once per trace in the event set.

Examining each event involves comparing the event against up to t events contained in $S_{back}(E)$ to check if it is a predecessor. We will therefore check c valid events against t events, up to another c invalid events against t events, and up to one invalid event at the end of each path (maximum of n paths) against t events. Overall, the number of operations required for this computation is $O(ct + nt + t)$. The final t term is required as $O(t)$ operations are required for step 1 (initialization). This can be rewritten as $O(cn + n^2)$ since $n \geq t$.

For the existing algorithm then, the total number of operations required is $O(e + t^2 + nt^2 + ct + nt + t)$ or more simply, $O(cn + n^3)$ since $n \geq t$ and $c \geq e$.

4.3 New Algorithm

Next we present a new algorithm for computing a convex closure. We describe the algorithm below, comparing it to the existing algorithm where appropriate. We then prove its correctness and again derive the asymptotic execution time.

4.3.1 Description of New Algorithm

The new algorithm begins in a manner similar to the existing one; however, it incorporates several algorithmic changes that significantly improve the algorithm's speed. Intuitively, the approach to computing the *back* vector is similar to the existing algorithm, except that rather than checking all greatest predecessors of each event in T_{back} (Definition 4.4) against events in T_{front} (Definition 4.5), we compute the set of greatest predecessors of events in T_{back} (using Definition 3.12) and examine only this single set of greatest predecessors (to determine which are successors of T_{front}).

The approach to computing the *front* vector is completely different from the existing algorithm. Rather than walking forward through events across multiple traces, we walk backwards on single traces individually as shown in Figure 4.2. This allows us to take advantage of the vector timestamps (which was not possible in the existing algorithm) as the timestamps contain information about events in the past rather than events in the future. Also, rather than comparing each event we encounter as we walk backwards on a certain trace against all events in T_{front} , we skip any events in T_{front} that we have already discovered not to be a predecessor of a future event on this trace. Additionally, once we find an event in T_{front} that is a predecessor, we can stop examining other events in T_{front} and immediately walk backwards one more event on the current trace.

In pseudocode, the new algorithm computes the *back* vector of E using the

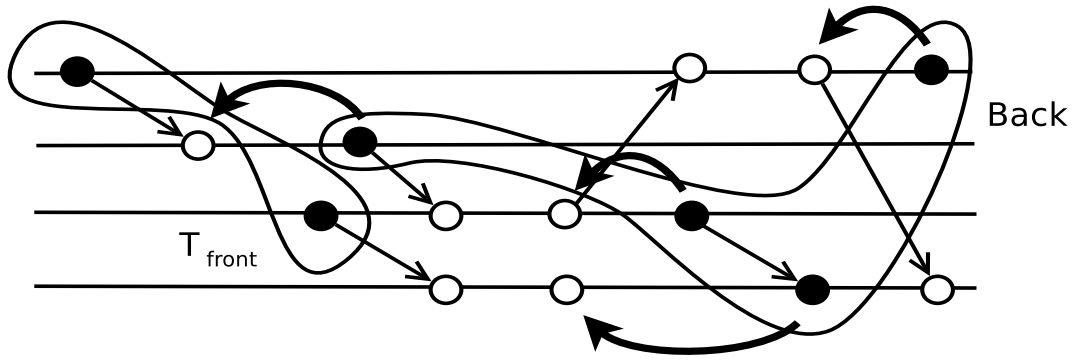


Figure 4.2: New Computation of the *front*

following steps. For this algorithm, an event's greatest predecessor on its trace is itself.

Algorithm 4.8. *New Computation of the Back Vector*

```

// First, find the least and most recent events in
// event set E on each trace
Tfront[1..N] = {null, ..., null}
Tback[1..N] = {null, ..., null}
for event in E
    if Tfront[event.trace] == null || event → Tfront[event.trace]
        Tfront[event.trace] = event
    if Tback[event.trace] == null || Tback[event.trace] → event
        Tback[event.trace] = event
// Having computed Tfront and Tback, we next examine the greatest
// predecessors (GP) of the events in Tback (see Theorem 4.1)
// We only store a GP if we have not seen
// a more recent GP on that trace previously
// Note that when this algorithm is complete, we have a superset of
// the back. Once the algorithm to compute the front is run, the back
// will be adjusted to the correct set of events
back[1..N] = {null, ..., null}
for i = 1 to N
    for j = 1 to N
        if Tback[i] != null && Tback[i].GP[j] != null
            && Sfront[k] → Sback[i].GP[j]
            && (back[j] == null || back[j] → Tback[i].GP[j])
                back[j] = Tback[i].GP[j]

```

The second half of the new algorithm computes the *front* vector of E using the steps below. Note that e_{trace} is the number of the trace that e is located on.

Algorithm 4.9. *New Computation of the Front Vector*

```

// Initialize event, i,
// q (the index of the event in Tfront we are comparing against)
// and front
i = 0
event = back[i]
q = 1
front[0 .. N] = {null, ..., null}
while true
    // Find the next event in back
    while event == null
        if i == N exit
        i++
        event = back[i]
    // If the event does not happen after the event we are
    // currently examining in Tfront then move to the next
    // event in Tfront
    // If we have exhausted Tfront, move to the next element
    // in back
    while (q ≤ N && Tfront[q] ↗ event)
        q++
    if q > N
        if event == back[i]
            back[i] = null
        i++
        q = 1
    else
        // Update the front and move one
        // event to the left
        front[event.trace] = event
        event = event's predecessor on this trace

```

4.3.2 Correctness of New Algorithm

To show the correctness of the new algorithm, we need to show two things. First, we must show that the resulting convex closure does not leave out any events. Second,

we must show that it does not include any unnecessary events. We developed the following theorem and corollary to assist in demonstrating correctness of the new algorithm.

Lemma 4.10. *Let e, f, g be events in the event set. If $e \rightarrow f$ and $g \nrightarrow f$, then $g \nrightarrow e$.*

Proof. We will use a proof by contradiction—it is valid in this domain since for any two events i, j we must have that either $i \rightarrow j$ or $i \nrightarrow j$. First, assume $g \rightarrow e$. Since we are given that $e \rightarrow f$ and the transitivity operator holds for primitive events then $g \rightarrow e, e \rightarrow f \Rightarrow g \rightarrow f$. However, we are told that $g \nrightarrow f$, so we have a contradiction. Therefore, the theorem holds. \square

Corollary 4.11. *Let e, f, g be events in the event set. If $e \rightarrow f$ and $e \nrightarrow g$, then $f \nrightarrow g$.*

Recall that the goal of the closure algorithm is to find all events, c , where $a \rightarrow c \rightarrow b$ and a, b are events in the input data set. Using Lemma 4.10, we can state that if an event does not happen before any event in T_{back} then it does not happen before any other event in the input data set. We also claim that the set of greatest predecessors of the set T_{back} is a superset of the *back* vector. By transitivity, any predecessor of an element in the input event set is also a predecessor of an element in T_{back} , thus we need only find predecessors of T_{back} and not predecessors of all events in the input event set.

Finally, we can remove any element from this set of greatest predecessors if it is not a successor of an element of T_{front} . By Corollary 4.11, if an element is not a successor of T_{front} it cannot be a successor of any other element in the data set. If it is a successor of T_{front} then it should be part of the *back* vector as it happens both after an event in T_{front} and before an event in T_{back} (unless it is an event in T_{back} ,

in which case we already require it be part of the closure). This set of arguments proves that the *back* vector is computed correctly.

When computing the *front* vector, we need only to examine traces that are present in the *back* vector and only events that happen before events in the *back* vector (for obvious reasons). We only need to determine, when examining an event, whether the event is a successor of an element in T_{front} . (We already know it is a predecessor of an element in T_{back} .) If we examine most-recent elements on each trace first and progress towards elements that are less recent, then by Lemma 4.10, we can stop once we find a single event that is not a successor of anything in T_{front} . Similarly, by Lemma 4.10, we can stop examining events against a *particular* event in T_{front} once we move back to an event that is not a successor of that particular event in T_{front} . Clearly, every event we encounter on a given trace (before we find such an event that is not a successor of anything in T_{front}) should be included in the convex closure as it must be a successor of something in T_{front} and a predecessor of something in T_{back} . This proves that the computation of the *front* vector is correct.

4.3.3 Asymptotic Behaviour of New Algorithm

As in the existing algorithm, the new algorithm to compute the back vector starts, in step 1, by examining all the events in the input set—in total, e events to determine T_{front} and T_{back} . Step 2 then computes the set of greatest predecessors over the set of events in T_{back} . (Because timestamps have already been computed for each event in T_{back} , the greatest predecessors of each event are obtainable with no extra work, due to Theorem 4.1.) Computing the complete set of greatest predecessors for T_{back} requires $O(nt)$ operations since there are t events in T_{back} having at most n greatest predecessors each. Overall, the improved back-vector computation requires $O(nt + e)$ operations.

When computing the front vector, at each step we either increment q or examine a new event. (Note that q is never decremented. It is only reset to 0 after we are finished examining a given trace.) We stop looking at a given trace once we encounter an event that is not part of the convex closure. Given this, we will examine at most $c + n$ unique events (c successful and n unsuccessful) and will examine an event more than once on, at most, nt occasions (q will be incremented from 0 up to t , the number of events in T_{front} , for each trace in $GP(E)$). Thus, the total number of operations for computing the front vector is $O(nt + c + n)$.

The entire improved computation takes $O(nt + c + n + e)$ operations. Since $n \geq t$ and $c \geq e$ we can express this more simply as $O(n^2 + c)$ operations. Compared to the existing computation, we have improved performance by a factor of n , the number of traces in the complete event data set. Additionally, the constant factors in the new computation are smaller than in the original and this is demonstrated in the empirical results since the operations in both algorithms are straightforward and there are no hidden constants.

4.4 Additional Improvements

Additional changes we made to the new closure algorithm to further improve its performance are discussed in this section. The improvements are mentioned here, rather than in the description of the new algorithm, for clarity. All additional improvements center around the computation of the *front* vector. When determining the *front*-vector event for a given trace, we start by looking at the event in the *back* vector for that trace and then move left, toward less-recent events. As originally described, the algorithm moves left only one event each time. If we instead move by more than one event each time, we can improve the algorithm's performance.

One approach involves using a straightforward binary-search technique. We can

choose the initial left value for the binary search to be the first event on the trace and the initial right value as the event on that trace in the *back* vector. A possible problem with this approach is that it does not take into consideration the non-uniform costs of evaluating different events. The cost to verify that an event should not be part of the closure is much larger than the cost of verifying that an event is in the closure. There are, however, other costs involved when computing a convex closure related to the number of events examined (such as computing or fetching timestamps) and since the binary-search technique minimizes the number of events we examine, it is worth evaluating. The final section of this chapter compares the performance of the binary-search technique with a new technique we describe next.

Recall from the description of the new algorithm that the cost of evaluating the events that end up in the closure for a specific trace is equal to the number of events in the closure on that trace plus t , the number of events in T_{front} . The cost of evaluating against events in T_{front} is, in effect, amortized over all the events in the closure from that trace. Unfortunately, when we evaluate an event not in the closure, the cost of evaluating events in T_{front} is borne each time. In each case we have to check against events in T_{front} that have indices from q up to $t - 1$ and q remains the same for the next evaluation. Given this non-uniform cost, we would like to tailor our “step-back” technique so it is more likely we will examine an event that is in the closure than one that is not.

Another possible “jump-back” strategy behaves as follows. First, let i be the number of events on the current trace from the start of the trace to the event on the trace in T_{back} and let t be the number of events in T_{front} . Our *initial* jump-back amount will be set to $x = i/t$. Clearly, we can do at most t jumps back, using this initial value, and we will have an amortized cost of t checks against T_{front} over all events discovered to be in the closure plus the single event we encounter outside the closure. (We always stop upon finding an event that is outside the closure.)

The cost of these initial jumps is $O(t)$. Once we do encounter an event outside the closure, we know the boundary between closure and non-closure events lies somewhere between the last event we discovered to be in the closure and the event we just discovered to be outside the closure. Further, we know that the number of events between these is exactly x , since that was our jump amount.

For the next iteration, we will decrease our jump amount to $y = x/t$. Again, we will incur a cost of $O(t)$. In total, we can decrease our jump amount $\log_t i$ times (since we divide our jump size by t each time). Given this, the total cost of jumping in this fashion *per trace* will be $O(\log_t i * t)$. Thus, the total time required by this approach is $O(\log_t i * t * n)$.

Now, let us compare this runtime to the $O(nt + c)$ runtime of the single-stepping approach. A casual inspection of these two runtimes will show that the size of c relative to n and t will, in large part, determine which approach is faster. Let $t = n^k$, where $0 < k \leq 1$ (as $t \leq n$ and we will assume that $t > 1$). Furthermore, let $c = n^j$. We can rewrite the runtime of the jump-back approach as $O(\log_{n^k} i * n^{k+1})$. We can also rewrite the runtime of the single-stepping approach as $O(n^{k+1} + n^j)$.

Given these two runtimes, we can make some observations. First, if $j > k + 1$, then the single-stepping approach will have a dominant term with a higher power (unless i is *extremely* large relative to n^k , such as $i = n^{k^{n^x}}$). Second, if k is small (for example, $k < 0.1$), and j is not much larger than $k + 1$, then the single-stepping approach will likely have a faster runtime, as the constant on the jump-back runtime will be quite large and the power on the dominant term of the single-stepping approach will be only slightly larger than that on the jump-back approach. Finally, if both j and k are large, then the power on the dominant term of the single-stepping approach will far outweigh the modest constant in the runtime of the jump-back approach as $\log_{n^k} i$ decreases quickly as k increases.

Although we have not provided a definitive way to decide which algorithm should be used, the performance-evaluation section of this chapter presents an empirical analysis of the three approaches (the single-step, binary-search, and jump-back methods) that provides further insight as to which approach should be used in a given situation.

The second, and final, improvement is also related to the same part of the algorithm, but is simpler. If there is an event in T_{front} on the same trace as the initial event e , from the set of greatest predecessors, then we can immediately skip back through all events between the event in T_{front} and e , since we know that all those events are successors the event in T_{front} (and obviously predecessors of e). We then continue the algorithm as usual, starting with the immediate predecessor of T_{front} on the same trace.

4.5 Combining Convex Closures

Sometimes we would like to build a convex closure with events contained in two existing closures. That is, we would like to combine two closures into one. This section will explore some ways we can do this efficiently.

The most straightforward way to combine two closures involves simply extracting the events from each original closure, and then passing the combined sets of events to a closure algorithm. The set returned from the algorithm is the combined closure. This approach is valid as $CC(CC(A) \cup CC(B)) = CC(A \cup B)$, where A and B are the input event sets to the two original closure operations. With this approach, however, we are discarding most of the work done in creating the original two closures.

Although previous work on combining closures exists [9, 63], the method proposed in the previous work only provides more efficient ways to combine closures in

certain limited situations. The method requires that at least one of the two closures contain only a single event, that the greatest predecessors of that event are a subset of the greatest predecessors of the back of the other set, and that the single event happens after the other set. Unfortunately, there are many situations where at least one of these conditions does not hold and in that case, we must revert to the straightforward approach mentioned above. In such a case, we end up doing extra work (to determine if the proper conditions hold) and thus, the closure takes longer than if we had used the straightforward approach to begin with. Our improved approach, as detailed below, describes how we can more efficiently compute the convex closure given any two original closures as input.

If we focus on the information gathered during the course of the original two closure operations, we can take some simple, but significant shortcuts that can noticeably improve the speed of computing the combined closure. In particular, the two original computations each produce a *front* and *back* vector and a set of greatest predecessors of each T_{back} .

First, knowing the *front* and *back* of each original closure allows us to compute T_{front} and T_{back} efficiently. We examine the two fronts and two backs and record the most-recent event and least-recent event per trace. Because the *front* and *back* vectors from the original closures tell us the most-recent and least-recent events per trace from each closure, it is unnecessary for us to examine any other events within the original closures to compute T_{front} and T_{back} for the new closure.

Next, we can also re-use the original sets of greatest predecessors of T_{back} from each of the original closures (although we must remember to store this information when computing a closure). If we take the most-recent event per trace over these two sets of greatest predecessors, it will give us the set of greatest predecessors of T_{back} for our new, combined closure. In fact, since we can compute the greatest predecessors of T_{back} for our new closure in this way without examining the contents

of the new T_{back} , it is unnecessary to compute the new T_{back} in the previous step of the algorithm.

Finally, we can take greater advantage of the second improvement mentioned in Section 4.4. Since we already computed a *front* event for each *back* event in our original closures, using this improvement means that we can jump back such that all events included in the original closures will be included in the new closure with no additional work required. Only events to be added to the new closure that were not included in either of the original closures will need to be examined by the algorithm. Since we, in effect, use the least-recent events from both sets as the *front*, and the most-recent events from both sets as the *back*, we maximize the benefit of this particular optimization.

The final section of this chapter will provide several empirical results that compare the three methods above: the straightforward approach, the approach developed by Bedassé and Ward [9, 63], and the new approach.

4.6 Test Setup

We describe the test setup used to perform the tests presented in the next section (as well as those in Chapters 5 and 6). We use the Java source code (packages `poet.core`, `poet.model`, and `poet.ui`) as well as Eclipse (3.3 or later) [2] and GEF (3.3 or later) both available at <http://eclipse.org>. We also use a database. (Both `hsqldb` [3] and `mysql` [4] are supported.) Once Eclipse and GEF have been installed, the source-code packages can be imported. When the import has completed and the code has compiled, we can run a second instance of Eclipse containing our plugin that allows us to import existing data sets, view imported data sets on the screen, and search imported data sets for a given pattern. (Before doing so, we need to start our chosen database and ensure it is listening on a port for connections.) This

second instance can be started by choosing **Run**→**Open Run Dialog...** and then double clicking on “Eclipse Application” to create a “New Configuration” as shown in Figure 4.3. Clicking “Run” will start the second instance and will automatically include any plugins inside source-code packages in the workspace.

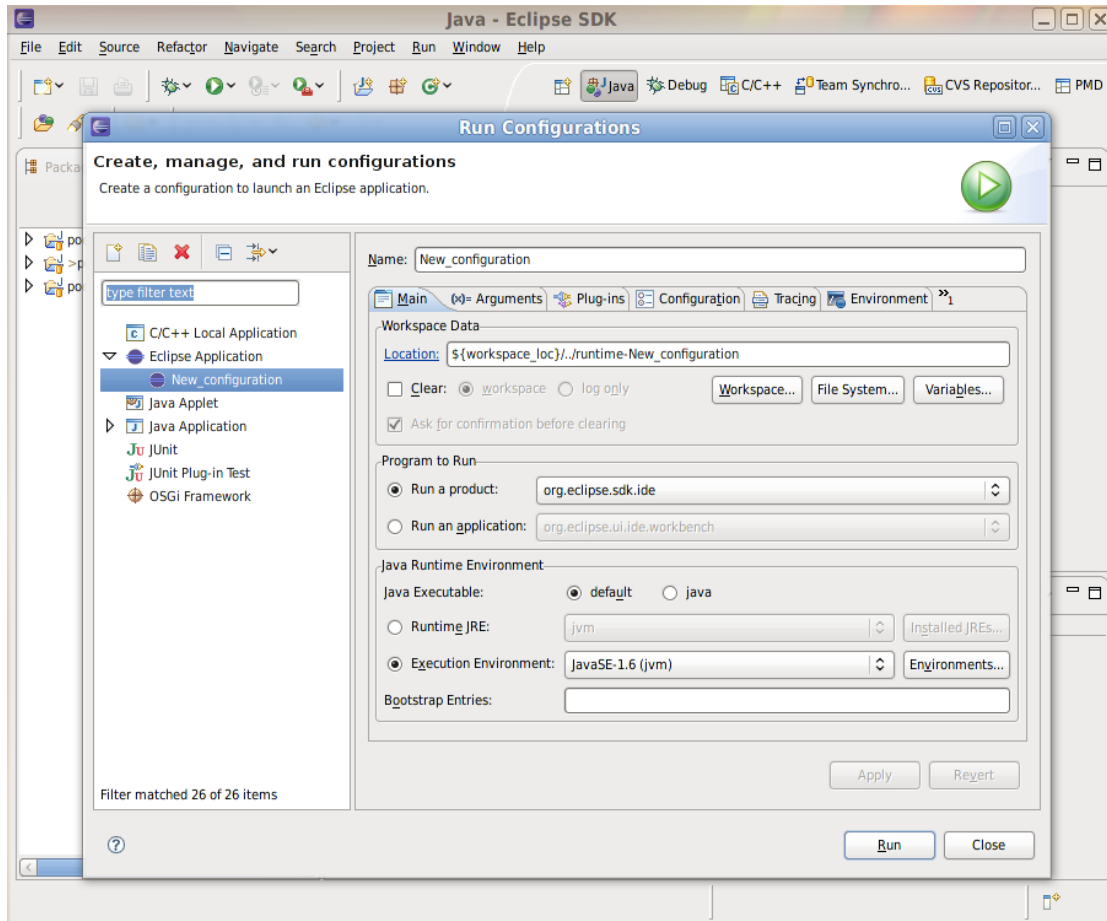


Figure 4.3: Starting a Second Instance of Eclipse

Once the second instance of Eclipse has been started, our first step should be to import an existing partial-order data set. Originally, it was only possible to import data sets created by the C-based POET implementation [37, 55, 56], but recent work now allows a wider variety of data sets to be imported [50], including Java profiling data collected by TPTP [5]. Every imported data set must be placed inside a project, so our first step is to create a new project. Clicking **File**→**New**→**Project...** and then selecting “Project” under the “General” group

and then following the steps in the wizard will accomplish this.

After this empty project folder has been created, we can import data sets into this project. This is done by right-clicking the newly-created project, and choosing **New**→**Other...** and finding “Event Database” under the “POET” heading (shown in Figure 4.4). The wizard will prompt for various pieces of information. All of the existing event-database (.uef) files are located in the “poet.model” project in the “data” folder. (The next section will describe the various event databases used for our testing and will provide the filename of each.) The .tgt_descr files, which describe the environments from which the data was collected, are also located in this same “data” folder.

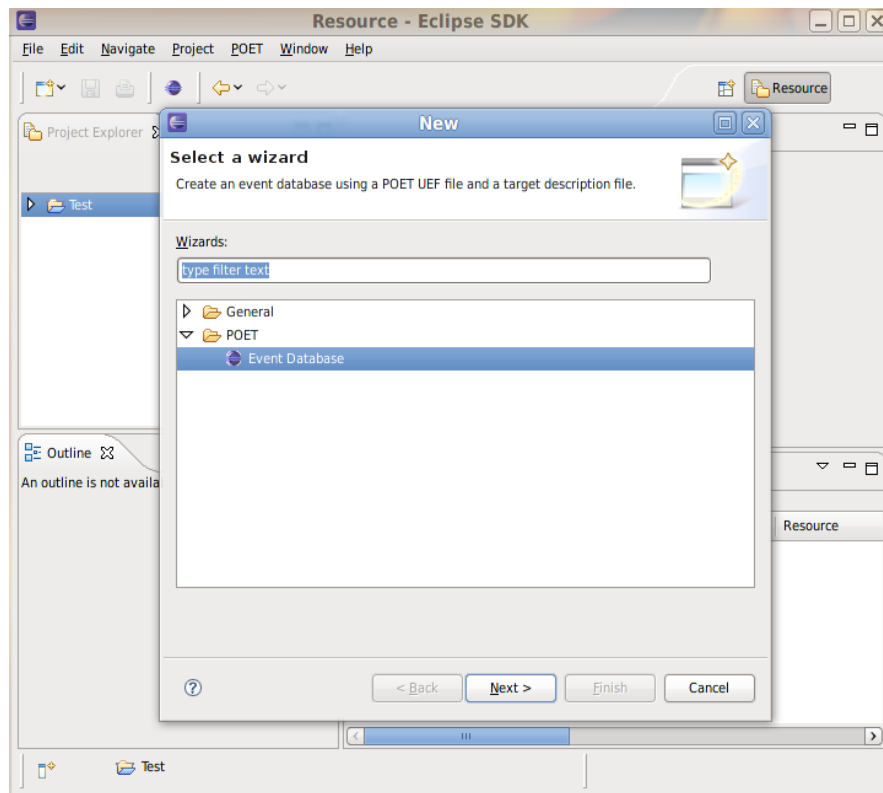


Figure 4.4: Importing a Partial-Order Data Set

Now that we have imported a data set, we can execute the various tests in Chapters 4, 5, and 6. Each test is run on the command line (but pattern search can also be done interactively by opening an imported data set and clicking **POET**→**Find**

Event Patterns). The command-line invocations below can be used to run the various tests. Note that \$wkspc is the location where the Eclipse workspace containing the source code is located and the following paths should be added to the normal Java classpath: \$wkspc/poet.core/bin, \$wkspc/poet.model/bin, \$wkspc/poet.ui/bin:, \$wkspc/poet.model/lib/hsqldb.jar, and \$wkspc/mysql-connector-java-5.0.8-bin.jar. The specific argument values for the command-line invocation and the flags that need to be set in the code to run each test are described in the sections providing the respective test results.

Invocation Name	Invocation
Searcher	java poet.core.pattern.Searcher _\$1_\$2. \$3 \$4 \$5
Random	java poet.core.ConvexUtil _\$1_\$2.

4.7 Performance Evaluation

Although asymptotically, the new algorithm is faster, it is also important to measure the difference in performance in practice. To assess the real-world performance of the existing algorithm compared to the new algorithm, we evaluated the algorithms using two methods. The first method uses eleven pattern-based test cases and the second method computes convex closures for sets of random events and also combines some of these random closures.

Each pattern-based test case was composed of a different pattern matched against one of four sets of event data. We measured, for each of the eleven test cases, the elapsed time to discover all matches to the pattern for each of the two algorithms.

Test cases were run on event data collected from the TCP-socket, $\mu C++$ [10, 11], and PVM [38] environments. The TCP-socket data set is described Section 3.5 and

more details about the first five test cases can be found there as well. The data set contains 303 events, most of which are data transfers, over 8 traces.

Our first test case uses the “FirstConnectionEstablished” pattern from Example Pattern 1 shown in Figure 3.10. It attempts to discover the first successfully established connection (there may be multiple concurrent connections that meet that criterion). Our second test case uses the “LastConnectionEstablished” pattern, also based on Example Pattern 1. It attempts to discover the last successfully established connection. The third test case used is based on the patterns “FinalDataTransferC1” and “FinalDataTransferC2” from Example Pattern 2 shown in Figure 3.11. Only one of these two patterns will return a match as we are attempting to discover the final data transfer made to the server. We sum the time taken and number of event comparisons over both patterns. (One returns a single match and one returns no matches.) Our fourth test case uses patterns “ConsecutiveC1Sends” and “ConsecutiveC2Sends” from Example Pattern 3 shown in Figure 3.12. The test case is designed to discover all pairs of consecutive send operations on each of the two clients. We again sum the results from the searches on each client. The fifth and final test case for this data set is based on the “HappensBeforeC1C2” pattern from Example Pattern 4 shown in Figure 3.13. This case verifies that all data transfers from client to socket are followed by a data transfer from socket to server.

The second data set used in this performance evaluation was collected from a $\mu C++$ environment. $\mu C++$ is an extension of C++ that incorporates concurrency constructs. The $\mu C++$ data set used for the test cases was gathered from a $\mu C++$ program containing an intentional bug. The program contains two methods that are protected by a single semaphore (at most one of the two methods should have a thread executing in it, and there should never be more than one thread in a given method). The intentional bug is that there is a 1% probability that the semaphore will not be acquired properly when a thread attempts to run one of the methods.

```

EnterMonitor1 := ["M1(0x0x9ac5730)", "thread received", ""];
EnterMonitor2 := ["M1(0x0x9ac5684)", "thread received", ""];

ConcurrentMonitors := EnterMonitor1 || EnterMonitor2; (Case 6)

ThreadStart := ["", "thread start", ""];
ThreadStop := ["", "thread stop", ""];
ANY := ["", "", ""];

StartStop := (ANY --> ThreadStart) --> ThreadStop; (Case 7)

IndThreads := (ThreadStart-->ThreadStop) ||
               (ThreadStart-->ThreadStop); (Case 8)

```

Figure 4.5: μ C++ Patterns

The result is that there may be two or more threads accessing the given methods simultaneously. Since each of the 5 threads tries to execute the methods 2000 times and there are 4 events each for acquiring and releasing semaphores as well as entering and exiting the protected methods (as well as others such as start/stop thread events), the data set contains 177,735 events over 11 traces. Patterns six through eight operate on this data set and are shown in Figure 4.5.

Our sixth test case checks for the bug we introduced. Specifically, it checks whether “method A is entered” is concurrent with “method B is entered”. The algorithms each discover all 65 such occurrences. In all of the test cases used, the two algorithms also located identical sets of matches.

Our seventh test case verifies that “start thread” and “stop thread” events are recorded correctly in the data set. Specifically, the test checks for any events that occur before the “start thread” event (there should be only a “create” event in each case) and also checks that “start thread” precedes “end thread”. Since there are interactions between threads, each of the “start thread” events will precede multiple “end thread” events, but examining the data will quickly allow us to pick out the matches we are interested in. Specifically, this test case first finds events where an

```

Send := ["", "send", ""];
Recv := ["", "recv", ""];
ANY := ["", "", ""];

ConSend := (Send || Send || Send || Send || Send ||
            Send || Send || Send || Send ); (Case 9)

SendSend := (Send -(ANY)-> Send); (Case 10)

SendRecv := (Send -(ANY)-> Recv); (Case 11)

```

Figure 4.6: PVM Patterns

“ANY” event precedes a “thread start” event and then where the compound event we just discovered, “ANY” + “thread start”, precedes a “thread end” event.

The eighth test case (the final one for this data set) searches for concurrent traces in the μ C++ data set; that is, it attempts to find pairs of traces that have no interaction with each other. None of the thread traces from the previous case should be concurrent with each other. The only traces that should be concurrent are various control traces that do not interact with each other, either paired with each other or with one of the thread traces from case 1. There are 78 such pairs of traces.

The third and fourth data sets were collected under the PVM environment. The third set is a simulation of a distributed merge-sort operation containing 138 events across 16 traces and the fourth data set is a simulation of the game Life under PVM containing 5898 events across 48 traces. The simulation starts with one node sending data to all the other nodes. Each node then chooses a neighbour to relay the data to. Whenever a node receives data from one of its neighbours, it forwards it on to another neighbour. The patterns in this data share some of the characteristics of peer-to-peer file sharing networks. Patterns nine through eleven operate on these data sets and are shown in Figure 4.6.

The ninth test case operates on the PVM binary-merge data and searches for 9 concurrent send operations. Given the nature of the data, we know that there should never be an occurrence of 9 concurrent send operations, since at most half of the nodes are sending data concurrently. Thus, no matches are found for this pattern. (If we search for 8 concurrent send operations, then we discover a match.)

Our tenth test case operates on the PVM Life data set and searches for two consecutive send operations. This can be done using a pattern containing a single limited operator (i.e. $-(ANY)->$) and specifying that the event class on the limited operator is “ANY” as two events being consecutive implies there are no events of any kind between them. The goal of this test case is to verify that a node always sends out data immediately after receiving it and then waits to receive more data before sending again. The only time there should be two consecutive send operations is when the first node is sending out initial data to the other nodes.

The eleventh and final test case operates on the PVM Life data set and searches for all sends that are followed immediately by receives. This, in effect, allows us to count the number of successful data transfers that occurred over the lifetime of the simulation.

Case	Average Time Elapsed			Matches Found
	Existing	New (Binary Search)	New with Closure Combining	
1	48 ms	46 ms	39 ms	1
2	54 ms	49 ms	42 ms	1
3	323 ms	211 ms	172 ms	1
4	1669 ms	1257 ms	1088 ms	58
5	244 ms	174 ms	135 ms	0
6	90,947 ms	87,931 ms	87,463 ms	65
7	15,412 ms	13,522 ms	13,555 ms	28
8	51,456 ms	4,614 ms	4,624 ms	78
9	479 s	435 s	374 s	0
10	9427 s	9119 s	9287 s	47
11	9445 s	9522 s	9441 s	1918

Table 4.1: Pattern-Based Performance Analysis, Part 1

Case	Timestamp Comparisons			Matches Found
	Existing	New (Binary Search)	New with Closure Combining	
1	552	3137	1249	1
2	780	4392	1752	1
3	46,008	73,270	40,567	1
4	2,253,970	3,454,520	1,919,459	58
5	54,694	82,710	45,511	0
6	37,466,097	37,466,878	37,465,448	65
7	1,422,817	1,425,327	1,424,711	28
8	35,051	122,543	92,249	78
9	$1.037 * 10^9$	$1.164 * 10^9$	$0.725 * 10^9$	0
10	$1.453 * 10^9$	$1.549 * 10^9$	$1.477 * 10^9$	47
11	$1.493 * 10^9$	$1.611 * 10^9$	$1.528 * 10^9$	1918

Table 4.2: Pattern-Based Performance Analysis, Part 2

Tables 4.1 and 4.2 display, respectively, the total real time time and total number of timestamp comparisons needed by the pattern-search algorithm to discover all matches to a pattern when using the existing convex-closure algorithm, the new convex-closure algorithm, and the new convex-closure algorithm with closure combining. For these tests, we use the binary-search method with the new convex-closure algorithm as we will see later in this section that it performs about the same as, or slightly faster than, the “jumpback” method and is much faster than the step-back-by-1 approach. We also use the final improvement mentioned in Section 4.4 for the new closure algorithm (with and without closure combining). The tests were run on a Core 2 Quad CPU at 3.0 GHz with 6 GB of RAM under the Fedora Core 8 distribution of Linux. The underlying database used was MySQL version 14.12. Each time in the table is the average over six test runs and the range of results for each set of test runs was within 3% of the average value over those test runs.

To run each test, we first followed the steps described in Section 4.6. The old algorithm was tested by setting the flag “oldCC” to true in `poet.core.ConvexUtil.java`. The new algorithm (using binary search, but no closure combining) was tested

by setting the flags “oldCC”, “jumpBackSearch”, “original_closure_combining”, and “new_closure_combining” to false, and “binarySearch” to true. (These flags are all located in `poet.core.ConvexUtil.java`.) To test the new algorithm with closure combining, we assign the same values to the flags as in the previous set of tests, except that “new_closure_combining” is set to true. In all three cases, after setting the appropriate flags, we run the tests using the “Searcher” command from Section 4.6. We set argument \$1 to the name we gave the project into which we imported the data set, \$2 to the name we gave the data set, \$3 to the name of the pattern file in which we typed our pattern, \$4 to the name of the pattern we want to search on (e.g., “ConSend”), and \$5 to “noflat”.

The results indicate that for these test cases, the new convex-closure algorithm with closure combining is fastest, from between 5% faster and ten times as fast as the existing algorithm. The difference in the number of timestamp comparisons is not as dramatic (and sometimes the existing algorithm requires many fewer comparisons). This is a direct result of the algorithmic behaviour of the existing algorithm as it uses partner information to traverse forward through event data rather than relying solely on timestamp comparisons, as the new algorithm does.

The case that results in the most significant increase in speed is case 8. Since this case involves taking the closure of the start and end events on a trace, it involves a large number of events. The existing algorithm must walk forward through all of these events; whereas, the new algorithm can jump back quickly through the large number of events in the manner of a binary search.

It is also interesting to note the difference in runtime when we use closure combining with the new algorithm. Applying closure combining usually results in an improved runtime or an equivalent runtime (within the margin of error) relative to not using closure combining. Closure combining works best for search patterns that require large convex closures. In the final two test cases, for example, all

the closures involve only two events. In cases 8 and 9, most of the closures are performed on subpatterns that, again, only involve two events. By nature, closure combining can do little to speed up closures of this size.

In addition to the pattern-based tests, we ran the convex-closure algorithms alone (rather than as part of a pattern search) on random sets of events of various sizes. We ran one set of tests on each of the four data sets mentioned previously. We compare the existing algorithm to three versions of the new algorithm: the new algorithm using the default step-back-by-1 method, the new algorithm using binary search, and the new algorithm using the “jumpback” approach described in Section 4.4.

Tables 4.3 through 4.10 display the minimum, average, and maximum elapsed time to compute a set of convex closures as well as the number of event comparisons done in each case. We also show the results in Figures 4.7 through 4.10 with the error bars representing a 95% confidence interval (assuming a normal distribution). For each set size, either a single value, n , or a range, n_1 to n_2 , we chose either 5000 or 100 random sets of size n , or of size between n_1 and n_2 , events from the respective data set. We then measured how long, in total, it took to compute these 5000 (or 100) closures. To get the minimum, average, and maximum values in the tables, we repeated these tests 20 times. Thus, the minimum value shows the fastest computation of 5000 (or 100) closures out of the 20 runs, and the maximum value shows the slowest computation of 5000 (or 100) closures out the 20 runs. Again, the tests were run on a Core 2 Quad CPU at 3.0 GHz with 6 GB of RAM under the Fedora Core 8 distribution of Linux. The underlying database used was MySQL version 14.12. To run each test, we again started by following the steps described in Section 4.6 (unless we had previously done so). This time, we do not need to set any flags in the code. We run the tests using the “Random” command from Section 4.6 and set argument \$1 to the name we gave the project into which we

n	Time Elapsed (ms)/Comparisons per 5,000 Closures							
	Existing				New, Step Back by 1			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	39	46	176	87,818	17	17	21	201,273
4	65	69	76	145,926	34	35	44	425,156
6	84	84	89	171,717	39	40	44	537,003
8-10	98	99	103	186,223	43	44	48	598,707
15-20	115	116	120	199,481	41	42	46	613,942

Table 4.3: Random-Set Analysis for TCP Event Set, Part 1

n	Time Elapsed (ms)/Comparisons per 5,000 Closures							
	New, with Binary Search				New, with Jumpback Search			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	15	18	22	179,962	15	15	16	182,126
4	25	25	29	325,132	25	26	29	332,584
6	28	28	32	403,272	28	29	32	410,688
8-10	30	30	34	456,150	31	32	35	463,842
15-20	31	31	36	495,008	32	32	36	501,361

Table 4.4: Random-Set Analysis for TCP Event Set, Part 2

imported the data set and \$2 to the name we gave the data set. Note that due to the long runtimes for the $\mu C++$ event set, we perform only 100 closures (versus 5000 for the other data sets).

Looking at the results, we see that again, although the number of timestamp comparisons for each of the new algorithms is higher than for the existing algorithm, the new algorithms are noticeably faster, especially as the size of the input set increases. The differences among the algorithms are more prominent in these tests

n	Time Elapsed (ms)/Comparisons per 100 Closures							
	Existing				New, Step Back by 1			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	1787	1929	2072	3283	617	687	795	2,053,451
4	1648	1749	1850	3355	1310	1529	1999	4,259,598
6	3617	3717	3818	3626	1378	1498	1612	4,522,336
8-10	4232	4275	4318	3713	1303	1471	1625	4,423,495
15-20	4684	4695	4707	3753	1101	1264	1364	3,732,836

Table 4.5: Random-Set Analysis for $\mu C++$ Event Set, Part 1

n	Time Elapsed (ms)/Comparisons per 100 Closures							
	New, with Binary Search				New, with Jumpback Search			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	6	6	7	12,547	7	7	7	14,044
4	7	7	7	14,611	7	7	7	16,696
6	6	6	6	15,601	6	6	7	18,087
8-10	6	6	7	16,677	6	6	7	19,246
15-20	5	5	6	18,414	6	6	7	20,559

Table 4.6: Random-Set Analysis for $\mu\text{C++}$ Event Set, Part 2

n	Time Elapsed (ms)/Comparisons per 5000 Closures							
	Existing				New, Step Back by 1			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	22	23	27	50,321	11	11	16	197,197
4	45	45	47	192,090	19	21	35	389,860
6	65	66	71	348,949	27	28	33	544,194
8-10	91	91	95	574,467	34	35	40	712,055
15-20	137	138	144	1,015,210	44	45	50	993,174

Table 4.7: Random-Set Analysis for PVM Binary-Merge Event Set, Part 1

n	Time Elapsed (ms)/Comparisons per 5000 Closures							
	New, with Binary Search				New, with Jumpback Search			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	10	14	17	197,711	10	10	15	197,647
4	17	19	40	392,455	17	18	22	389,947
6	23	23	27	546,671	23	24	28	542,142
8-10	29	29	34	712,105	29	30	34	708,006
15-20	35	37	42	984,212	38	38	44	986,344

Table 4.8: Random-Set Analysis for PVM Binary-Merge Event Set, Part 2

n	Time Elapsed (ms)/Comparisons per 5000 Closures							
	Existing				New, Step Back by 1			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	220	234	326	389,785	117	127	135	1,077,294
4	647	661	701	1,259,334	482	495	509	3,468,517
6	1076	1094	1107	2,206,494	852	874	986	5,816,500
8-10	1640	1655	1692	3,483,266	1312	1345	1399	8,918,948
15-20	2714	2765	2909	6,060,937	2088	2116	2179	14,328,323

Table 4.9: Random-Set Analysis for PVM Life Event Set, Part 1

n	Time Elapsed (ms)/Comparisons per 5000 Closures							
	New, with Binary Search				New, with Jumpback Search			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	86	89	94	889,728	88	91	98	901,990
4	205	211	240	2,194,726	215	222	250	2,208,554
6	309	315	346	3,470,550	330	337	341	3,420,657
8-10	418	423	429	5,087,442	470	475	481	4,945,771
15-20	540	545	554	8,016,637	675	683	691	7,883,205

Table 4.10: Random-Set Analysis for PVM Life Event Set, Part 2

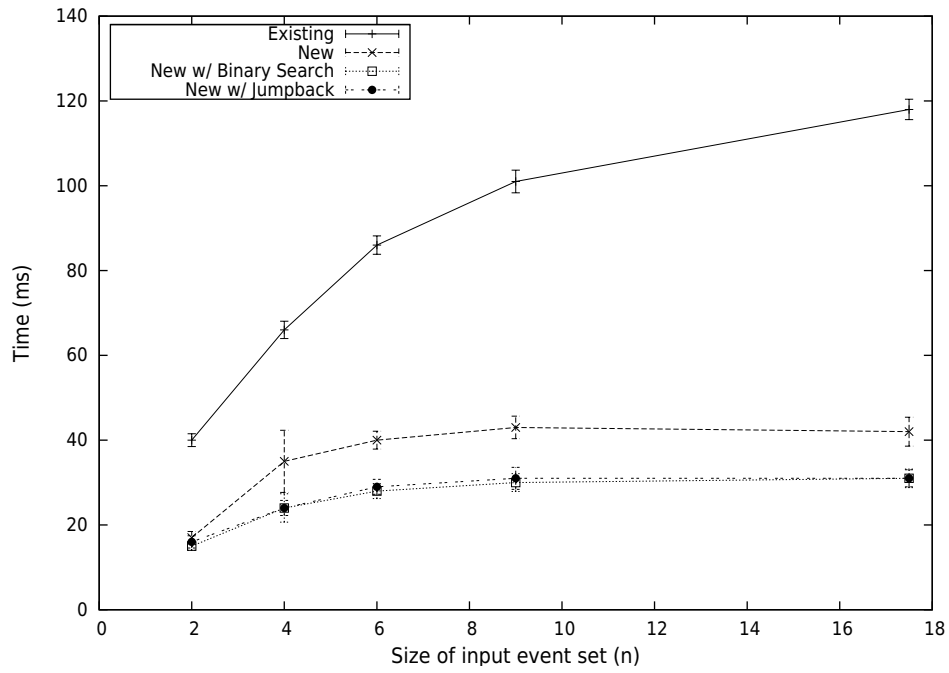


Figure 4.7: Graph of Random-Set Analysis for TCP Event Set

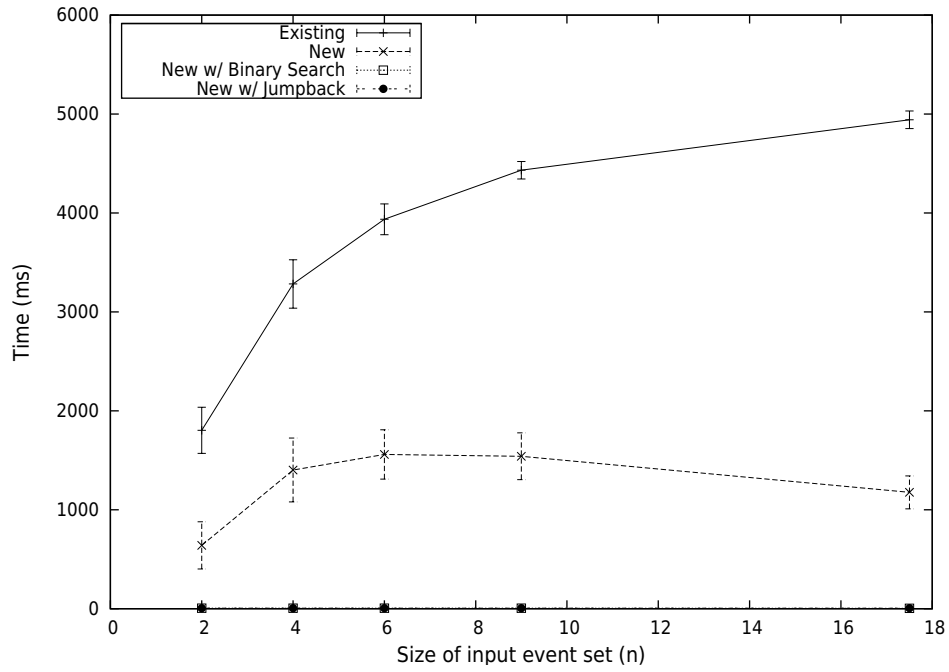


Figure 4.8: Graph of Random-Set Analysis for $\mu C++$ Event Set

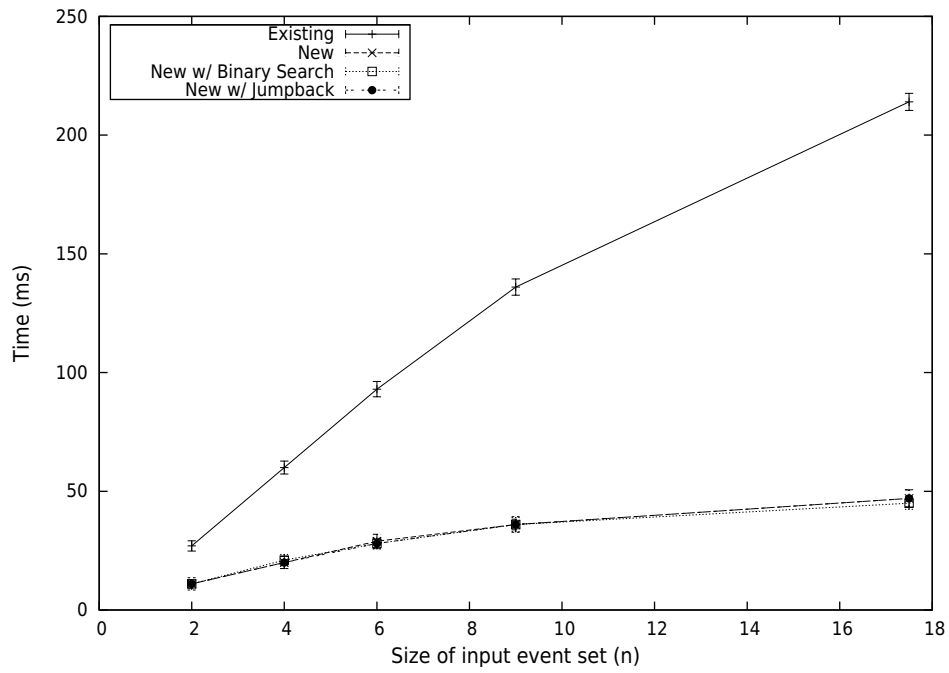


Figure 4.9: Graph of Random-Set Analysis for PVM Binary-Merge Event Set

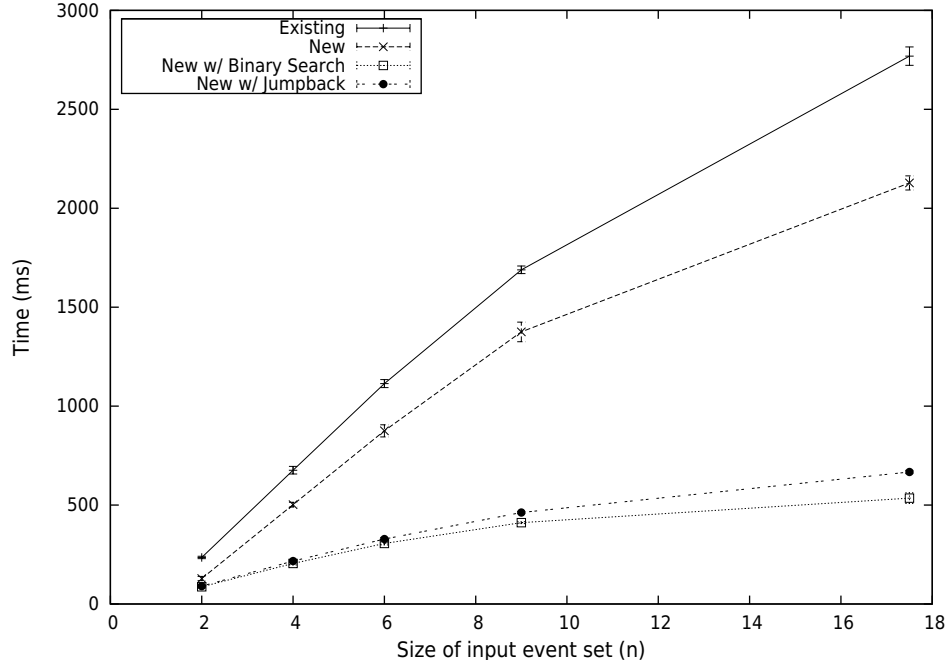


Figure 4.10: Graph of Random-Set Analysis for PVM Life Event Set

as compared to the first tests as we are not involving the other parts of the pattern-matching algorithm that tended to offset the differences somewhat in the earlier tests.

On average, the best versions of the new algorithm (the binary search and jumpback approaches) are around three times faster than the existing, but in one case, the difference is about three orders of magnitude. It is also interesting to compare the step-back-by-1 approach, the binary search approach, and the jumpback approach. The first is often noticeably slower, but is sometimes equivalent in speed to the other two. The other two are generally the same speed, although in the final test case, the binary-search approach seems to do better for larger closure sizes than the jumpback approach. Since the binary-search approach is never bettered by any of the other algorithms (within the margin of error), we will use only the binary-search approach for the remaining tests involving the new closure algorithm.

Our final set of tests evaluates the performance of the new convex-closure-

n	Time Elapsed (ms)/Comparisons per 5,000 Combines							
	No Closure Combining				Closure Combining			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	20	21	27	187,792	11	11	12	83,250
4	23	25	35	226,983	10	11	20	84,860
6	25	25	34	224,437	9	9	10	72,799
8-10	25	25	33	218,501	9	9	10	61,934
15-20	24	25	32	211,909	9	9	9	52,282

Table 4.11: Closure-Combining Analysis for TCP Event Set

combining algorithm (using the binary-search approach). The tests are run under the same setup as the previous random-based tests. We also have a variety of set sizes and we used the same number of test runs and methodology as in the previous tests. Our goal is to measure how long it takes to build the combined closure of two random closures of the given set size. In each case, the column n indicates the size or range of sizes for each of the two closures being combined. Before beginning the tests, we pre-compute random convex closures of the given size so that the test itself only involves the time required to combine the two closures. For the “no closure combining” method, the combined closure is built by taking the union of the events in the fronts and backs of the two closures and computing a new closure based on these events. The “closure combining” method is the technique described in Section 4.5.

The results are shown in Tables 4.11 through 4.14. Figures 4.11 through 4.14 show the results with error bars representing a 95% confidence interval (assuming a normal distribution). Note that this time we perform 5,000 closure combines for the $\mu\text{C++}$ test case. Also, note that we do not show the results for the closure-combining method developed by Bedassé and Ward [9, 63] as it provided less than a 1% improvement in most cases, and less than a 5% improvement in the best case. Such improvements are not significant as they are well within the 95% confidence interval of the approach that uses no closure combining.

n	Time Elapsed (ms)/Comparisons per 5,000 Combines							
	No Closure Combining				Closure Combining			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	64	69	94	278,152	33	36	58	58,173
4	62	68	86	278,246	31	31	33	58,216
6	60	60	62	278,591	29	30	52	58,249
8-10	58	59	82	278,736	27	27	28	58,611
15-20	53	54	75	279,398	23	23	25	59,351

Table 4.12: Closure-Combining Analysis for $\mu\text{C++}$ Event Set

n	Time Elapsed (ms)/Comparisons per 5,000 Combines							
	No Closure Combining				Closure Combining			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	17	17	18	223,403	10	11	24	60,850
4	27	27	28	419,078	12	13	21	94,608
6	30	31	39	535,830	13	14	15	94,803
8-10	34	35	43	626,635	13	13	14	84,279
15-20	36	38	46	688,074	12	12	13	61,584

Table 4.13: Closure-Combining Analysis for PVM Binary-Merge Event Set

n	Time Elapsed (ms)/Comparisons per 5,000 Combines							
	No Closure Combining				Closure Combining			
	Min	Avg	Max	Comp.	Min	Avg	Max	Comp.
2	124	132	141	2,142,401	101	105	115	938,965
4	286	292	303	6,049,156	222	227	237	2,766,457
6	404	418	492	8,546,010	306	311	323	3,936,751
8-10	476	487	502	9,932,135	354	361	374	4,513,492
15-20	462	474	549	9,770,578	324	332	345	4,020,494

Table 4.14: Closure-Combining Analysis for PVM Life Event Set

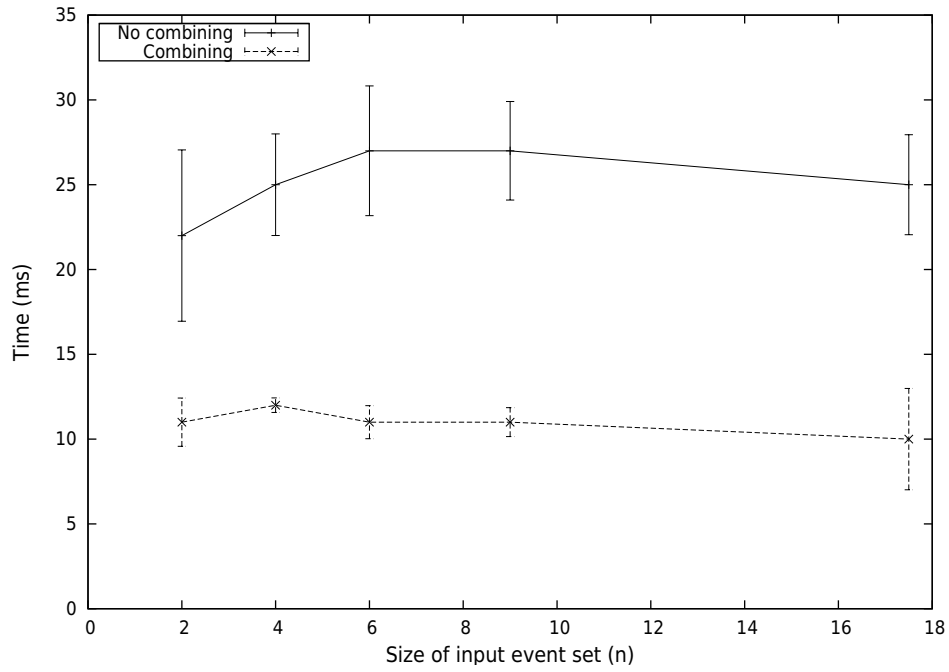


Figure 4.11: Graph of Closure-Combining Analysis for TCP Event Set

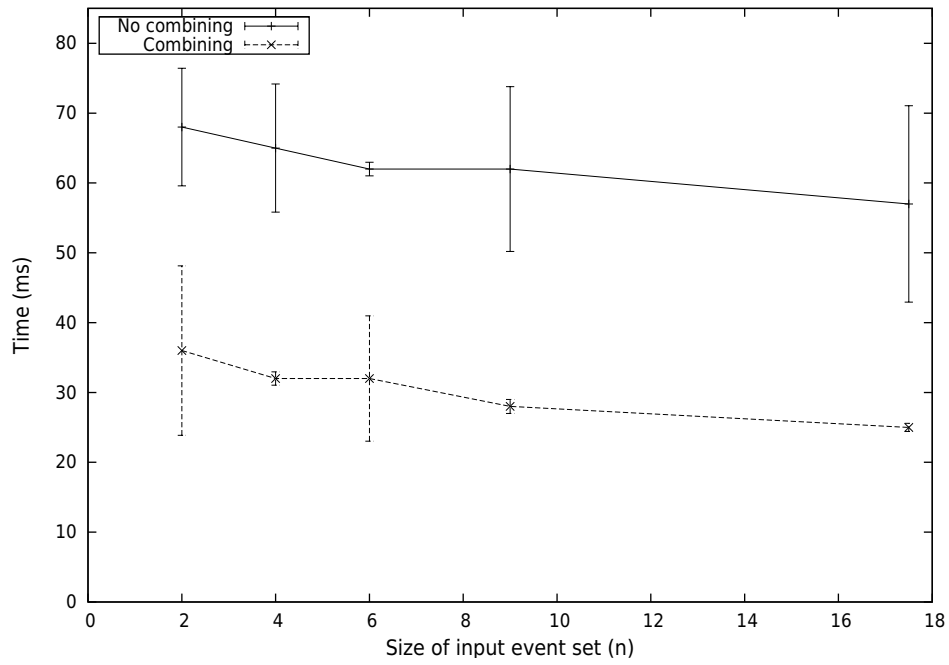


Figure 4.12: Graph of Closure-Combining Analysis for $\mu\text{C}++$ Event Set

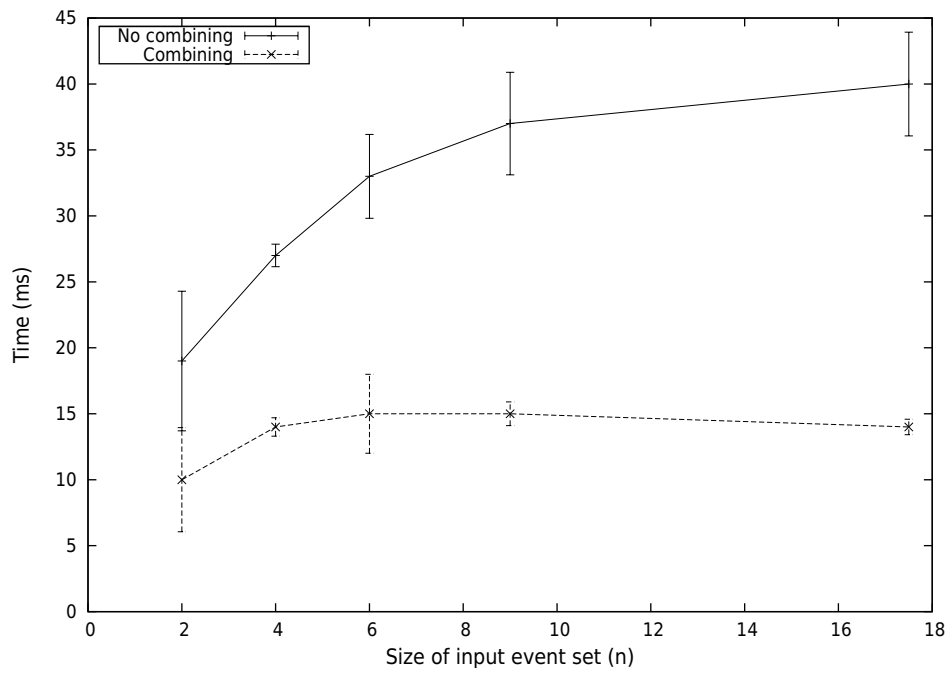


Figure 4.13: Graph of Closure-Combining Analysis for PVM Binary-Merge Event Set

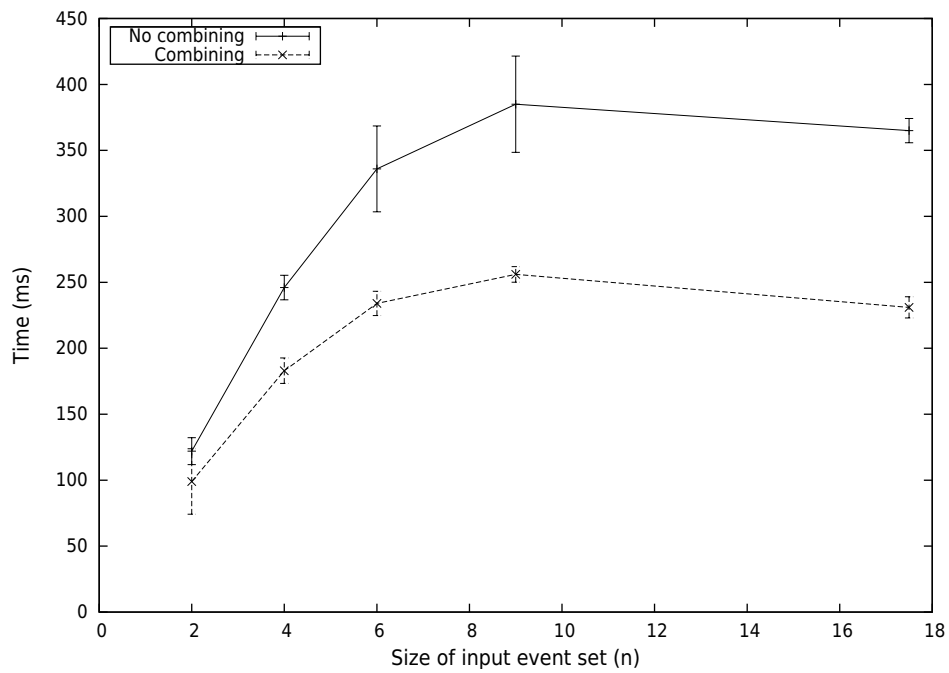


Figure 4.14: Graph of Closure-Combining Analysis for PVM Life Event Set

Looking at these results, we can see that in practice, there is a noticeable increase in performance when using closure combining. Unlike previous tests, since the two algorithms use the same fundamental approach, the differences in time elapsed corresponds more closely to the differences in timestamp comparisons. The closure-combining method has more overhead and so some of the gains made through fewer comparisons are lost in this overhead. Typically, there appears to be a 1.5 to 3 times difference in speed.

It is interesting to note that increasing the size of the input event set does not always increase the time required to compute a closure using the new algorithm in both the random-set and closure-combining tests. The most significant reason for this phenomenon is the final improvement from Section 4.4 that allows us to skip over all events between the event we are examining and an event in T_{front} that is on the same trace. As the size of the input event set approaches the number of traces in the event set, the percentage of traces in the resulting closure that are represented in T_{front} will increase and will give us greater leverage from this improvement.

We demonstrated, through the results above, that the new closure algorithm is superior to the existing algorithm. We also showed that the binary-search approach is slightly faster than the jump-back approach and that the new closure-combining method provides some additional gains.

Chapter 5

Pattern Rewriting

BECAUSE OUR original approach to pattern search requires the computation of a convex closure at each non-leaf node of the tree and this closure is non-trivial to compute (even with the improvements of the previous chapter), the original search algorithm can be slow. Rather than computing the convex closure explicitly at each step in the pattern search, we incorporate extra elements into the pattern itself that enforce convexity. In effect, we are expanding the original pattern by applying the definitions of the various compound-event operators from Chapter 3 directly rather than relying on computations of convex closures and comparisons of the resulting convex-event timestamps.

This new approach allows us to leverage knowledge about the relationships among events contained in the pattern. For example, if the search pattern were simply $(\mathcal{A} \parallel \mathcal{B}) \rightarrow \mathcal{C}$, the original algorithm would pass the two primitive events matching \mathcal{A} and \mathcal{B} to the convex-closure algorithm and the convex closure of those two events would be calculated before determining if they precede the event matching \mathcal{C} . The new algorithm leverages the fact that once a valid match is found, we know that the two events matching \mathcal{A} and \mathcal{B} (a and b respectively) are concurrent, and thus there can be no intervening primitive event, z , such that $a \rightarrow z$ and $z \rightarrow b$.

In such a case, no extra work would be required to ensure convexity of $(\mathcal{A} \parallel \mathcal{B})$.

Below, we explain how patterns are rewritten and how the predicate-detection algorithm is modified to search using these rewritten patterns.

5.1 Potential Benefits of Rewriting

The goal of rewriting patterns is to improve the efficiency of the pattern search. The most obvious, and direct, benefit of rewriting is to eliminate the need to perform a costly convex-closure operation at each step in the pattern search. The other benefit is that, after rewriting, the pattern is no longer in a rigid hierarchical structure and the various components of the rewritten pattern can be reordered. Also, after returning a match, it is simpler to restore the previous search state when processing a rewritten pattern, relative to a hierarchical one.

5.1.1 Eliminating the Convex Closure

The original purpose of computing the convex closure of a compound event was to ensure that a compound event, E , was never compared against another event that simultaneously happened both before and after E . The convex-closure algorithm discovers all such intervening events and adds them to the original compound event. Since the convex-closure-search approach does not consider non-disjoint matches to be valid, adding these intervening events to the original compound event ensures that they will never be compared (as part of another event) against the original compound event.

The disadvantage of the above strategy is that, in many cases, we will never attempt to compare the intervening events against the original compound event and so the effort in computing the convex closure is wasted. Indeed, in some cases,

we conclude from a careful analysis of the pattern that it is impossible that such a comparison will ever be attempted. For example, in the pattern $\mathcal{A} \parallel (\mathcal{B} \rightarrow \mathcal{C})$, it is easy to see that any event, a , matching \mathcal{A} can never be an intervening event of the compound event, $b \rightarrow c$, matching $(\mathcal{B} \rightarrow \mathcal{C})$ since $a \nrightarrow c$ (and also $b \nrightarrow a$).

Rather than attempting to discover all intervening events, a more efficient approach is to check, only when necessary, whether specific events are intervening events. We can do this by applying the definitions of the compound-event operators from Chapter 3 directly. For example, if we are searching for the event pattern $(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow (\mathcal{C} \rightarrow \mathcal{D})$ and we have discovered primitive events a, b, c, d that match event classes $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ then we only need to check that c and d are not intervening events in the compound event $(a \rightarrow b)$ and that a and b are not intervening events in the compound event $(c \rightarrow d)$. These checks usually require much less computation than discovering all intervening events in each of $(a \rightarrow b)$ and $(c \rightarrow d)$. Adding the extra conditions present in the convex-event definitions to the pattern itself during the rewriting stage allows us to eliminate the computation of the convex closure at each step in the search process.

5.1.2 Reordering and Pruning Pattern Components

In addition to eliminating costly convex-closure computations, pattern rewriting facilitates further improvements to the efficiency of pattern search. In the original search algorithm, there was limited flexibility with respect to the order in which nodes could be evaluated—a parent node could not be evaluated until both its children were evaluated. Using rewritten patterns and the new predicate-detection algorithm offers much more flexibility. The disjunctions in the rewritten pattern can be evaluated in any order and the happens-before relationships within disjunctions can be re-arranged as well.

Rewriting a pattern in the manner described here will usually make it substantially larger; however, it is usually the case that many terms can be eliminated from the rewritten pattern as they are only true when one or more of the other terms in the pattern are true. For instance, if our rewritten expression were $(\$a \rightarrow \$b) \wedge (\$a \rightarrow \$c) \wedge (\$b \rightarrow \$c)$ then by transitivity, we could eliminate the third happens-before relationship because if the first two expressions are true, then the third will be true. Chapter 6 discusses these matters further and provide some preliminary empirical results that compare the effectiveness of various techniques for optimizing the rewritten pattern.

5.2 A Basis for Rewriting

Our goal in rewriting a pattern is to eliminate the hierarchical nature of the original pattern and express it using only a minimal set of operators. In the following sections we show how patterns can be rewritten such that the resulting pattern

- has, at most, a 2-level hierarchy that is a conjunction of disjunctions of happens-before relations,
- uses only \wedge , \vee , \rightarrow , \rightarrow , $=$, and \neq operators,
- contains only event classes or variables representing event classes as the two operands to the \rightarrow operator, and
- allows \sim and $*$ modifiers on variables

The process of rewriting patterns takes full advantage of the formal definitions of all operators presented in Chapter 3 as they pertain to compound events. These definitions describe how to evaluate precedence between any two compound events,

but it may not be immediately clear how we can leverage these to allow us to rewrite patterns.

First, we examine how patterns are evaluated using the convex-closure approach from the previous chapter. To simplify matters, we assume, for now, that only the \rightarrow , \parallel , and \leftrightarrow operators are present in any pattern. (We will handle the \wedge , \vee , $!$, $*$, and \sim operators and modifiers later.) Convex events, just like primitive events, have timestamps associated with them. These timestamps allow us to evaluate the precedence relationships between any two convex or primitive events (or one of each). To evaluate a pattern, we must examine each operator and the left and right operands of the operator. If either operand is not a leaf node, we must evaluate the subtree first to determine that operand (which will be a convex event). When both operands satisfy an operator, we take the convex closure of the two operands and this becomes an operand for the parent operator. Once all the operators have been satisfied, we have found a match.

Now, how can we perform the same procedure, leveraging the compound-event operator definitions from Chapter 3 so that convex closures are unnecessary? If we examine a pattern statically, we see that it explicitly tells us which events match it and the relationships among those events. For example, if we find a match to the pattern $(\mathcal{A} \rightarrow \mathcal{B}) \parallel \mathcal{C}$, we know the match contains exactly three events: a , b , c that match event classes \mathcal{A} , \mathcal{B} , and \mathcal{C} respectively. This idea also applies to subpatterns. If we look at the larger pattern $((\mathcal{A} \rightarrow \mathcal{B}) \parallel \mathcal{C}) \rightarrow (\mathcal{D} \rightarrow \mathcal{E})$, we know that a match to the subpattern $(\mathcal{A} \rightarrow \mathcal{B}) \parallel \mathcal{C}$ again contains exactly three events that match the respective event classes. Regardless of which specific three events satisfy $(\mathcal{A} \rightarrow \mathcal{B}) \parallel \mathcal{C}$, we know that they alone satisfy that subpattern. We can then leverage this information about subpatterns to help us evaluate the “parent” operator.

In the example $((\mathcal{A} \rightarrow \mathcal{B}) \parallel \mathcal{C}) \rightarrow (\mathcal{D} \rightarrow \mathcal{E})$, assume our next step is to determine whether the three-event match that we have already determined satisfies the subpattern $(\mathcal{A} \rightarrow \mathcal{B}) \parallel \mathcal{C}$ happens before a two-event match that satisfies the subpattern $(\mathcal{D} \rightarrow \mathcal{E})$. Since we know that a , b , and c and d and e have already satisfied their respective subpatterns, we only remain concerned about the final operator that joins these two subpatterns. What we are really trying to determine is whether a compound event (the three-event match to the first subpattern) happens before another compound event (the two-event match to the second subpattern). We can make this determination by using the happens-before definition for compound events (Definition 3.7) which, in this case, expands as $\$a \rightarrow \$d \vee \$a \rightarrow \$e \vee \$b \rightarrow \$d \vee \$b \rightarrow \$e \vee \$c \rightarrow \$d \vee \$b \rightarrow \e (at least one event in the first compound event happens before at least one in the second), and $\$d \nrightarrow \$a \wedge \$d \nrightarrow \$b \wedge \$d \nrightarrow \$c \wedge \$e \nrightarrow \$a \wedge \$e \nrightarrow \$b \wedge \$e \nrightarrow \c (no event in the second compound events happens before any event in the first), and $\$a \neq \$d \wedge \$a \neq \$e \wedge \$b \neq \$d \wedge \$b \neq \$e \wedge \$c \neq \$d \wedge \$c \neq \e (no event in the first compound event is equal to any in the second). Note that $\$a$, $\$b$, $\$c$, $\$d$, and $\$e$ are variables corresponding to classes \mathcal{A} , \mathcal{B} , \mathcal{C} , \mathcal{D} , and \mathcal{E} respectively and are used so we can reference a particular bound value in multiple locations.

To expand the entire pattern (not only the final operator, as we just did above) we simply apply the definitions repeatedly starting at the leaves of the parse tree and moving towards the root. As long as our pattern contains only the \rightarrow , \parallel , and \leftrightarrow operators, then recursively applying the definitions will always yield a valid result. (We further argue this at the end of this section.) We showed in Chapter 3 that a convex event is equivalent to the original event set with respect to these three operators: whether we take the convex closure of each of $(\mathcal{A} \rightarrow \mathcal{B}) \parallel \mathcal{C}$ and $(\mathcal{D} \rightarrow \mathcal{E})$ and then determine whether the first closure happens before the second will yield an identical result to applying the compound-event happens-before definition directly.

Our goal is ensuring that each operator in the pattern is satisfied.

Further, we can express the resulting constraints as a single conjunction of disjunctions by using the same methods used to convert Boolean expressions into conjunctive normal form (CNF). For example, $a \rightarrow d \vee (a \rightarrow e \wedge b \rightarrow d)$ becomes $(a \rightarrow d \vee a \rightarrow e) \wedge (a \rightarrow d \vee b \rightarrow d)$.

Although it is straightforward to see how patterns containing only the \rightarrow , \parallel , and \leftrightarrow operators can be rewritten (based on the results of Chapter 3), we have yet to show how other operators and modifiers (\wedge , \vee , $*$, \sim) are evaluated properly. First, let us examine how the \vee operator is handled using the convex-closure method. Assume that patterns now can contain \rightarrow , \parallel , \leftrightarrow , and \vee operators. An \vee operator always has two operands, both of which are subpatterns. The \vee operator checks both subpatterns, starting with the left, and returns a convex event that satisfies one of the subpatterns. Rewriting patterns containing this operator (along with the \rightarrow , \parallel , and \leftrightarrow operators), is a straightforward process. We demonstrate how it is done by starting at the lowest levels of the parse tree. First, we find an \vee operator that has no \vee operators (from the original pattern) in any of its subpatterns. We know that the left and right subpatterns can be converted to CNF. Now, the convex-closure method would pick one of these (successfully-matched) subpatterns and pass it up the tree. Since the subpattern passed up varies depending on the data set, we must account for both possibilities in our static processing of the pattern. To do so, we simply take the “or” of the left and right subpatterns using standard Boolean techniques. For example, if the left subtree was rewritten as $(a \rightarrow d \vee a \rightarrow e) \wedge (a \rightarrow f \vee b \rightarrow d)$ and the right subtree as $(e \rightarrow f \vee g \rightarrow h)$, these would be combined as $(a \rightarrow d \vee a \rightarrow e \vee e \rightarrow f \vee g \rightarrow h) \wedge (a \rightarrow f \vee b \rightarrow d \vee e \rightarrow f \vee g \rightarrow h)$. If we examine this rewritten pattern, we see it is only satisfied when exactly one of the two subtrees is satisfied.

We generalize this approach as follows. Assume the left subpattern is $D_{1,1} \wedge D_{1,2} \wedge \dots \wedge D_{1,n}$ and the right subpattern is $D_{2,1} \wedge D_{2,2} \wedge \dots \wedge D_{2,n}$ where $D_{j,k}$ are disjunctions. For the \vee operator, we require that all the disjunctions from one of the two subpatterns be satisfied. When we rewrite the pattern at this step, the resulting pattern is $(D_{1,1} \vee D_{2,1}) \wedge (D_{1,1} \vee D_{2,2}) \wedge \dots \wedge (D_{1,2} \vee D_{2,1}) \wedge (D_{1,2} \vee D_{2,2}) \wedge \dots \wedge (D_{1,n} \vee D_{2,n})$. If every disjunction from one of the subpatterns is satisfied, then the rewritten pattern will be satisfied (since each disjunction in the rewritten pattern contains a disjunction from each of the two subpatterns). If even one disjunction from each of the two subpattern is *not* satisfied, then the rewritten pattern will not be satisfied, as there is a disjunction in the rewritten pattern containing only the single failed disjunctions from each of the two subpatterns.

We can then continue to rewrite \vee operators whose subpatterns have no \vee operators (from the original pattern) until the entire original pattern has been rewritten. A similar argument could be made for the \wedge operator.

Finally, we need to address how the $!$, \sim , and $*$ modifiers are rewritten. First, we will examine the \sim modifier. This modifier, if present on a variable, indicates that any matches to that variable should not be returned (not even to the parent of the current operator). For example, in the pattern $(\mathcal{A} \rightarrow \sim \mathcal{D}) || (\mathcal{B} \rightarrow \mathcal{E})$, we require that the event matching \mathcal{A} happens before the event matching \mathcal{D} ; however, when evaluating the concurrent operator, we do not consider the match to \mathcal{D} . As long as the matches for both \mathcal{B} and \mathcal{E} are concurrent with the match for \mathcal{A} , then the remainder of the pattern is satisfied. In general, when deciding which events make up the compound event for each of the left and right operands, we simply omit any events represented by variables marked with the \sim modifier unless the modifier has been applied to the entire left or right operand (and then we only consider it at the current level but not the next level up). For example, in the subpattern $(\mathcal{A} \rightarrow \sim \mathcal{D})$, we consider the event matching \mathcal{D} , but when evaluating the parent

(the concurrent operator) we do not.

Rewriting the $!$ and $*$ modifiers both require defining what it means to “negate” an expression. To demonstrate this, we show how to negate the CNF expression $(\$a \rightarrow \$d \vee \$a \rightarrow \$e) \wedge (\$a \rightarrow \$f \vee \$b \rightarrow \$d)$. We do so by applying De Morgan’s Theorem twice. The resulting pattern is $(\$a \nrightarrow \$d \wedge \$a \nrightarrow \$e) \vee (\$a \nrightarrow \$f \wedge \$b \nrightarrow \$d)$. Then, we can use techniques from earlier in this section to convert this expression to CNF. Because we can rewrite any pattern into a CNF expression, this approach gives us an effective technique to negate any pattern. For example, if we started with the pattern $(\mathcal{A} \rightarrow \mathcal{B}) \nrightarrow \mathcal{C}$, we could interpret this as $![(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow \mathcal{C}]$, then rewrite the expression in square brackets to produce a CNF expression and then continue by applying De Morgan’s Theorem. With the convex-closure approach there is no need for such complexity, as we can evaluate whether an operator is satisfied (returning a Boolean value) and then negate the Boolean value.

Although it is not immediately apparent, we also need negation when rewriting expressions modified with $*$. Using the convex-closure approach, evaluating these types of “for-all” expressions is straightforward. If our pattern is $clause_1 \rightarrow *(clause_2)$ then the happens-before operator is only satisfied if the convex event matching $clause_1$ happens before every convex event matching $clause_2$. (Note that the pattern language does not allow the $*$ modifier on clauses, only on variables representing clauses, but we do so here for simplicity. Since no examples in this section result in variables being bound across multiple clauses, we can make any of the examples in this section valid by replacing each clause modified with $*$ by a variable representing that clause.) When rewriting a pattern, this modifier poses a challenge as we would like to extract the elements from $clause_2$ and express the entire pattern in CNF. We must rewrite this particular modifier, otherwise hierarchical patterns like $clause_1 \rightarrow *(clause_2 \rightarrow *(clause_3 \dots))$ could exist. An examination of the semantics of the $*$ modifier facilitates devising a rewriting technique.

First, note that if the universally-quantified or “for-all” expression operand is *not* satisfied, then we are not concerned about its relationship with the other operand. As described in Chapter 3, when determining which values satisfy a “for-all” expression, we must iterate over all possible combinations of events matching the variables and event classes present in the expression. When evaluating the subpattern $*(\mathcal{A} \rightarrow \mathcal{B})$, for example, we need to evaluate the expression over all possible combinations of events matching \mathcal{A} and \mathcal{B} , and only the pairs of events that satisfy $\mathcal{A} \rightarrow \mathcal{B}$ are returned (as convex events) by this subpattern. Expressing this in a different way, the subpattern $*(\mathcal{A} \rightarrow \mathcal{B})$ returns all combinations of values for \mathcal{A} and \mathcal{B} , except those that satisfy $\mathcal{A} \nrightarrow \mathcal{B}$. This idea allows us to expand “for-all” subpatterns and move the $*$ modifier inside. If we have a pattern $expr_1 \rightarrow *(expr_2)$, our first step is to rewrite it as we would if we were given a happens-before operator with regular (not “for-all”) operands, but we also add “for-all” modifiers to each of the variables from the “for-all” operand. At this point, the rewritten expression requires that $clause_1$ happen before the entire state space of the variables in $clause_2$; however, we only want to require that $clause_1$ happen before the elements in the entire state space that actually satisfy $clause_2$, so we provide an alternative, namely, that the \rightarrow does not have to be satisfied for any elements in the state space that do not satisfy $clause_2$.

We illustrate this idea further with an example. If our pattern is $*(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow \mathcal{C}$, we first rewrite it as $*a \rightarrow \$c \vee *b \rightarrow \c (every event matching $\$c$ must be preceded by either all events matching $*a$, or by all events matching $*b$, plus, for simplicity, we have left out additional terms that check that the operands are not entangled), but that is overly restrictive, as it doesn’t consider the earlier restriction on $*a$ and $*b$. So, we can add an additional happens-before relationship to the expression and write it as $*a \rightarrow \$c \vee *b \rightarrow \$c \vee *a \nrightarrow *b$, which effectively means that if a given $*a$ and $*b$ combination does not satisfy $*a \rightarrow *b$, then we do not care about

its relationship to $\$c$. As a reminder, Chapter 3 states that “for-all” variables are bound over the entirety of a disjunction, not just for individual relationships in that disjunction.

This approach, as with the approaches used for the previous operators, allows us to incrementally convert a pattern into CNF by starting at the leaves and moving upwards. It ensures that the operands for every converted operator are in CNF and so the above mechanisms always receives input operands in a standard arrangement. The explanations above can be used to form a proof by induction as follows. We know the leaves are single variables or event classes and thus are trivially in CNF and thus can be properly rewritten (base case). We further know that every operator along the path from a leaf to the current operator takes two expressions in CNF and correctly returns an expression in CNF based on the techniques above (inductive step). Proving or reasoning about how each operator can take two CNF expressions and return a CNF expression that is equivalent to evaluating the operator using the convex-closure method is sufficient to prove that the entire original expression can be properly rewritten. At the evaluation of a given operator, we assume that everything up to this point (lower down the parse tree) has been correctly rewritten in CNF, and then show, as above, how we can rewrite the current operator correctly, given two CNF expressions as operands.

One final detail that was not mentioned is how to negate a happens-before expression containing a “for-all” variable in a disjunction within a CNF expression. One example of this type of expression is $\mathcal{A} \rightarrow * \mathcal{C}$. Again, reasoning about the semantics of the expression leads to an answer. If it is *not* the case that the event matching \mathcal{A} happens before all events matching \mathcal{C} , then it means that there is at least one match for \mathcal{C} that the event matching \mathcal{A} does not happen before. We can express this as $\mathcal{A} \nrightarrow \sim \mathcal{C}$.

The next section will briefly describe how this full conversion is done in practice.

5.3 Mechanics of Rewriting

Given a pattern, we require an algorithm that leverages the above rewriting techniques to rewrite the pattern, eliminating the need for costly convex-closure operations. Starting with a parse tree of the original pattern and an empty data structure for holding the rewritten pattern, we perform a post-order traversal of the tree and accumulate parts of the rewritten pattern as we go.

To apply the above techniques, we need to know the primitive events (or classes of primitive events) present in the left and right subpatterns or subtrees of each parent as well as each rewritten subpattern. Each leaf node represents a primitive event in the pattern and this information can be propagated up the tree as we traverse it, along with the rewritten elements we have generated thus far. This process will give each parent node knowledge of the full set of primitive events in each of its two subtrees and all the rewritten elements generated by the operators in its subtrees. Additionally, because each parent node stores its operator, we can apply the appropriate technique to produce a new rewritten pattern that combines the rewritten patterns of the two children with the current operator.

Finally, we need a simple, efficient representation of the rewritten pattern in CNF to minimize the amount of work that the modified predicate-detection algorithm needs to do when evaluating the pattern. Each rewritten pattern will be a single conjunction containing (possibly) multiple disjunctions and each disjunction will contain (possibly) multiple happens-before relationships that have a single variable or event class as each operand.

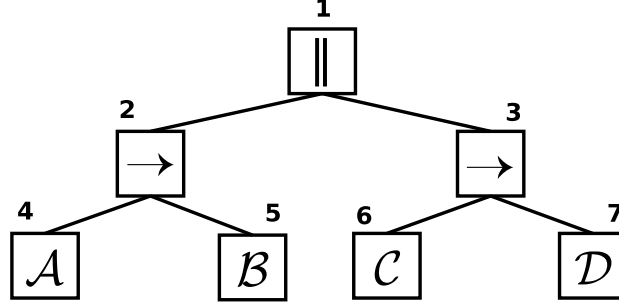


Figure 5.1: Pattern Parse Tree (Example 1)

5.4 Rewriting Examples

This section contains examples of how patterns can be rewritten using the strategy presented in the previous sections. Each example presents a pattern, shows the parse tree for that pattern, and demonstrates how the rewritten pattern is constructed.

Example 1: $(\mathcal{A} \rightarrow \mathcal{B}) \parallel (\mathcal{C} \rightarrow \mathcal{D})$

Figure 5.1 shows the parse tree of this pattern. As described in Section 5.3, we need to determine the classes of primitive events in the left and right subtrees of each parent and then apply the appropriate techniques from Section 5.2 to build the rewritten pattern. Before we start, we replace classes \mathcal{A} , \mathcal{B} , \mathcal{C} , and \mathcal{D} in the pattern by variables $\$a$, $\$b$, $\$c$, and $\$d$ respectively, as we need to reference them in multiple places in the final rewritten pattern.

We start with node 2 as nodes 4 and 5 are visited first and propagate up information to this node about the variables they specify. Since node 2 is a happens-before operator, we can rewrite it as $(\$a \rightarrow \$b)$. Next, we visit node 3 after nodes 6 and 7 are visited and propagate up their information to this node. Again, this node is a happens-before operator and again we can rewrite it as $(\$c \rightarrow \$d)$.

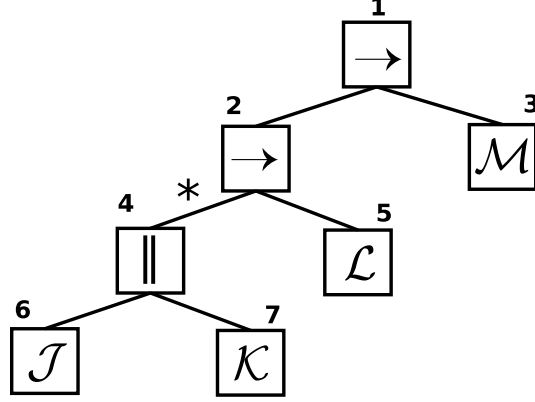


Figure 5.2: Pattern Parse Tree (Example 2)

Finally, we visit node 1. Node 2 has propagated up variables $\$a$ and $\$b$ that it has received from its children (along with any modifiers attached to those classes), as well as its current rewritten pattern, $(\$a \rightarrow \$b)$, and node 3 has propagated up variables $\$c$ and $\$d$ from its children as well as $(\$c \rightarrow \$d)$. Since node 1 is a concurrent operator, we apply the compound-event concurrent-operator definition (Definition 3.8) and add terms $(\$a \nrightarrow \$c)$, $(\$a \nrightarrow \$d)$, $(\$b \nrightarrow \$c)$, $(\$b \nrightarrow \$d)$, $(\$c \nrightarrow \$a)$, $(\$c \nrightarrow \$b)$, $(\$d \nrightarrow \$a)$, $(\$d \nrightarrow \$b)$, $(\$a \neq \$c)$, $(\$a \neq \$d)$, $(\$b \neq \$c)$, and $(\$b \neq \$d)$ to the rewritten pattern.

The complete rewritten pattern is the conjunction of the 14 terms above.

Example 2: $(*(\mathcal{J} \parallel \mathcal{K}) \rightarrow \mathcal{L}) \rightarrow \mathcal{M}$

Figure 5.2 shows the parse tree of this pattern. Again, we replace the event classes with variables and start by looking at an operator with leaf nodes as children.

Starting with node 4, we rewrite the concurrent operator as $(\$j \nrightarrow \$k)$ and $(\$k \nrightarrow \$j)$ and $(\$j \neq \$k)$ and pass this information up to node 2. At node 2, we have a universally-quantified expression as the left operand and need to apply the appropriate technique to rewrite the subpattern rooted at this node. We can

rewrite it as $((*j \rightarrow \$l \vee *k \rightarrow \$l) \wedge (\$l \nrightarrow *j) \wedge (\$l \nrightarrow *k) \wedge (\$j \neq \$l) \wedge (\$k \neq \$l)) \vee (*j \rightarrow *k \vee *k \rightarrow *j \vee *j = *k)$. Note that the expression in the first set of brackets, composed of 4 happens-before operators, is the standard expansion for a \rightarrow operator from the original pattern. The expression in the second set of brackets, composed of 3 happens-before operators, is the negation of the left universally-quantified operand. We can then rewrite the expression so it is a single conjunction of disjunctions by applying standard Boolean techniques. The resulting expression is

$$\begin{aligned}
& (*j \rightarrow \$l \vee *k \rightarrow \$l \vee *j \rightarrow *k \vee *k \rightarrow *j \vee *j = *k) \wedge \\
& (\$l \nrightarrow *j \vee *j \rightarrow *k \vee *k \rightarrow *j \vee *j = *k) \wedge \\
& (\$l \nrightarrow *k \vee *j \rightarrow *k \vee *k \rightarrow *j \vee *j = *k) \wedge \\
& (\$j \neq \$l \vee *j \rightarrow *k \vee *k \rightarrow *j \vee *j = *k) \wedge \\
& (\$k \neq \$l \vee *j \rightarrow *k \vee *k \rightarrow *j \vee *j = *k) \wedge
\end{aligned}$$

Finally, we visit node 1 that receives information from the left subtree (only about existentially-quantified $\$l$ and the rewritten pattern above) and from the right subtree (variable $\$m$). Because we are only concerned about one event class from each operand, only one disjunction needs to be added to the rewritten expression: $(\$l \rightarrow \$m)$. The complete rewritten pattern is the conjunction of these 6 disjunctions.

Note that in some cases a variable class will appear twice in the original pattern. To handle such a situation, we add subscripts to the variable replacing the class and increment the subscript on each appearance of the variable. For example $(\mathcal{A} \rightarrow \mathcal{B}) \rightarrow \mathcal{A}$ would first be rewritten as $(\$a_1 \rightarrow \$b) \rightarrow \$a_2$.

5.5 A Search Algorithm for Rewritten Patterns

The original search algorithm was designed to work on a pattern expressed as a tree structure where all the children of a parent node must be evaluated before the parent node itself. This was necessary as the convex closure of each child needed to be computed before the parent's node could be evaluated. In contrast to this, the rewritten pattern is a series of conditions that must be satisfied. Even though the rewritten pattern is no longer represented as a tree, the pattern search on the rewritten pattern can still be viewed as a CSP and a general backtracking approach still applies.

The first step the algorithm performs is obtaining a list of matches for a given event class or variable. This step only needs to be done once per event class (and each variable has an associated event class) as the other components of the pattern do not influence the set of events that match an event class. Next, the algorithm processes individual happens-before or equality expressions in the pattern. This is done by testing all possible combinations of values for each event class and determining if one happens before the other or if they are equal. It is possible that one or both of the variables in the happens-before expression have already been bound to an event by an earlier operation. In that case, only the single bound value for a given operand needs to be evaluated.

At yet another level up, the algorithm examines disjunctions and determines if a disjunction is satisfied. The algorithm starts by evaluating the first happens-before relationship in the disjunction. Once it finds a pair of values that satisfies it, it moves on to the next disjunction in the conjunction. When the algorithm eventually backtracks to this term, it will continue looking at the same happens-before relationship until it has exhausted all pairs of values that satisfy it. At that point, it moves on to the next happens-before relationship in the disjunction and

the algorithm will continue on as before. Once all happens-before relationships have been exhausted, the algorithm will backtrack to the previous disjunction in the expression.

If a disjunction contains universally-quantified, or “for-all,” variables, then special steps need to be taken. The disjunction must be true for all possible values assigned to the “for-all” variables. The set of “for-all” variables is given an initial assignment, which binds each “for-all” variable to a given event. If the disjunction is satisfied (by one of the happens-before relationships) then the “for-all” variables are given a new assignment and the disjunction is tested again. Once all possible combinations of values for the “for-all” variables have been tested, then we can move on to the next disjunction. Note that we could alternatively bind “for-all” variables across all disjunctions, but it is typically more efficient to bind for each individual disjunction as the “for-all” variables in each disjunction may be different. We require that each disjunction independently satisfy all combinations of the “for-all” variables as they are combined with \wedge operators, so this is a valid approach.

Finally, at the highest level, the algorithm determines if each disjunction has been satisfied and returns a match containing the appropriate events. The events matching every event class are returned as well as any events matching variables without $*$ or \sim modifiers. (Recall from Chapter 3 that these modifiers can only be placed on variables.) Whenever we discover a match, we typically want to display it to the user immediately rather than presenting a full list when the algorithm terminates; however, we would like to maintain the state of our implicit stack so we can quickly resume the algorithm once the user requests the next match. This is easily accomplished using multiple threads to simulate a coroutine. The search algorithm wakes a display thread, passes it the latest result and then puts itself to sleep after finding each match. When the user requests the next match, the display thread will then wake the search algorithm (so it can continue where it left off) and

then put itself to sleep.

To further illustrate how the search algorithm works, we will look at what happens when we search with the pattern $(\$j \rightarrow \$l \vee \$k \rightarrow \$l) \wedge (\$l \nrightarrow \$j) \wedge (\$j \rightarrow *m \vee *m \rightarrow \$j \vee *m \rightarrow \$k)$. (This pattern is used for convenience and is not necessarily rewritten from any original pattern, although it is in the correct format for rewritten patterns.) The search begins with the first disjunction, $(\$j \rightarrow \$l \vee \$k \rightarrow \$l)$, and looks at the first happens-before relationship, $(\$j \rightarrow \$l)$. Since $\$j$ and $\$l$ have not been assigned values yet, we assign each the first matches that satisfy their event classes. If we do not satisfy $\$j \rightarrow \l , then we move on to the next happens-before relationship in the disjunction (keeping $\$j$ and $\$l$ bound to their current values). In the next happens-before relationship, $(\$k \rightarrow \$l)$, we can only assign a value to $\$k$, as $\$l$ has been assigned already. If it is not the case that $\$k \rightarrow \l , then we continue trying new values for $\$k$ (since this is the final happens-before relationship in the disjunction). If we exhaust all possibilities for $\$k$, then we must backtrack to the previous happens-before relationship in the disjunction $(\$j \rightarrow \$l)$ and we choose the next value for $\$l$. Assume, this time, that $\$j \rightarrow \l is satisfied. In that case, we move on to the next disjunction (keeping the bound values for $\$j$ and $\$l$). We follow the same procedure for the next disjunction, $(\$l \nrightarrow \$j)$, although this disjunction is simpler because it has only one happens-before relationship. If we are able to satisfy it—clearly we will since $\$j \rightarrow \$l \Rightarrow \$l \nrightarrow \j —we move to the final disjunction $(\$j \rightarrow *m \vee *m \rightarrow \$j \vee *m \rightarrow \$k)$.

The final disjunction requires special care as it has a universally-quantified variable. Before we begin evaluating this disjunction, we assign a set of values to the universally-quantified variables (in this case there is only one: $*m$). Since $\$j$ is assigned, but $\$k$ is not, we choose an initial value for $\$k$. We begin verifying that the disjunction holds over all values of $*m$, starting with the initially-assigned value. If we fail to satisfy the disjunction for any value of $*m$, we must pick a new value for

k and again verify that the disjunction holds over all values of m . If we continue to fail and exhaust all values for k , we must backtrack. It is interesting to note that in some cases, a value for k may have already been assigned by the first disjunction (if we reach the second happens-before relationship). Thus, the way this disjunction is evaluated depends on the results from previous disjunctions.

If we succeed for all values of m , then we return the current match (the set of values currently assigned to the existentially-qualified variables). Since we are simulating a coroutine, as described above, we store the match in a shared variable, wake up the thread that reports the match to the user, and put ourselves to sleep. Once the next match is requested, the reporting thread will wake us up at the same state as where we left off and we continue trying the subsequent variable assignments.

Eventually, we exhaust all useful assignments to the existentially qualified variables and terminate the search.

5.6 Performance Evaluation

To assess the real-world performance of the original convex-closure-based search algorithm compared to the new rewriting algorithm, we evaluated the algorithms using eleven test cases. The test cases are the same as for Section 4.6, so we do not repeat the details of each case and event data set here. Table 5.1 shows, for each test case, the elapsed time to discover all matches to the pattern for each of the two algorithms. We also show the total matches found for each pattern and the total number of single-integer comparisons of timestamps performed for each algorithm.

Because we demonstrated in Chapter 4 that the convex-closure algorithm using the binary-search approach and the new closure-combining method was the fastest

way to execute the search, that method is the “original” against which we compare the “new” rewriting approach.

Case	Average Time Elapsed		Improve- ment	Matches Found	Timestamp Comp.	
	Original	New			Original	New
1	39 ms	20 ms	1.95 X	1	1249	644
2	42 ms	20 ms	2.10 X	1	1752	804
3	172 ms	42 ms	4.10 X	1	40,567	7672
4	1088 ms	189 ms	5.76 X	58	1,919,459	144,776
5	135 ms	87 ms	1.55 X	0	45,511	59,400
6	87,463 ms	12,001 ms	7.29 X	65	37,465,448	37,464,278
7	13,555 ms	8236 ms	1.64 X	28	1,424,711	13,349,356
8	4624 ms	4529 ms	1.02 X	78	92,249	3732
9	374 s	75 s	4.99 X	0	$0.725 * 10^9$	$0.508 * 10^9$
10	9287 s	281 s	33.05 X	47	$1.477 * 10^9$	$1.234 * 10^9$
11	9441 s	299 s	31.58 X	1918	$1.528 * 10^9$	$1.263 * 10^9$

Table 5.1: Performance Analysis for Pattern Rewriting

The tests were run on a Core 2 Quad 3.0 GHz CPU with 6 GB of RAM under the Fedora Core 8 distribution of Linux. The underlying database used was MySQL version 14.12. Each time in the table is the average over five test runs and the range of results for each set of test runs never varied more than 4% from the average value over those test runs. To run each test, we first followed the steps described in Section 4.6 (unless we had already done so for previous tests). The original algorithm (using binary search and closure combining) was tested by setting the flags “oldCC”, “jumpBackSearch”, and “original_closure_combining” to false, and “new_closure_combining” and “binarySearch” to true. (All are located in poet.core.ConvexUtil.java.) The new algorithm is unaffected by these flags. To execute the test cases, after setting the appropriate flags, we run the tests using the “Searcher” command from Section 4.6. We set argument \$1 to the name we gave the project into which we imported the data set, \$2 to the name we gave the data set, \$3 to the name of the pattern file in which we typed our pattern, \$4 to the name of the pattern we want to search on (e.g., “ConSend”), and \$5 to “noflat” (to

use the original algorithm) or “flat” (to use the new algorithm).

From these results we see that, in practice, there is a noticeable difference between the original and new algorithms. Even though the size of a rewritten pattern is larger, the new algorithm is, on average, about 10 times faster, depending on the pattern and event data set when compared to the convex-closure method. The number of timestamp comparisons needed by the new algorithm was lower in all but one case.

It is interesting to note that the difference in timestamp comparisons does not match up with the difference in speed of the two algorithms. Case 8 is a special example of this. The overhead involved in accessing the database in this case is approximately 4500 ms and this overhead can vary slightly between test runs. Thus, the time elapsed is not a good indicator of the difference between the two algorithms.

In general, there are some possible reasons for the overall discrepancy between timestamp comparisons and elapsed time. First, the new algorithm does little more than perform timestamp comparisons between events, so there is very little overhead. Conversely, the original algorithm needs to do a significant amount of traversing of the parse tree and needs to frequently compute convex closures. Even though the original algorithm uses the fastest convex-closure algorithm from Chapter 5, there is still much overhead compared to the new rewriting method and the overhead involves more than just timestamp comparisons. The smaller difference in timestamp comparisons may also indicate that there is a strong potential that optimizing the rewritten pattern will improve performance.

Chapter 6

Optimizing Rewritten Patterns

ALTHOUGH REWRITING alone decreases runtime, when compared to the convex-closure approach, it additionally provides opportunities to increase performance through various optimizations. This chapter introduces some optimization techniques and provides some empirical results demonstrating their potential.

Our optimizations involve modifying both the pattern and the search algorithm. We first examine how to eliminate redundant terms from a pattern and whether it always makes sense to do so. Next, we examine how to leverage information about the pattern and various metrics to allow us to rearrange portions of a pattern so that we fail (or succeed) quickly. This re-arrangement can be done either statically before we start the search (easier to do, but we have less data on which to base our decision), or dynamically during the search (harder to do, but we can use the results thus far to help us make a more informed decision). This chapter addresses only static re-arrangement. Lastly, we look at how the search algorithm can more efficiently process universally-quantified variables by maintaining a least-recently-used list.

Our overriding goal when doing these optimizations is to complete the pattern

search as quickly as possible. This equates to performing as few precedence evaluations as possible.

It is interesting to note that there are many similarities between reordering patterns and optimizing database queries (as first mentioned in Chapter 2). Primarily, both estimate the cost of ordering a query (the costs of various orderings of joins in a database query, for example) through use of metrics and statistics. Metrics for the lower-level portions of the query (e.g., metrics for a subquery performed on a single table that restricts rows based on indexed attributes) are much easier to estimate than those for a higher-level subquery that may join against several tables and restrict rows based on non-indexed attributes. The ability to do accurate cost estimation on a database query is still an open problem [12]. One implication of this is that there is currently no practical way to measure the strength of most optimizations, other than through empirical means.

6.1 Pruning

A rewritten pattern is usually larger than the original pattern. However, it is frequently the case that many happens-before relationships can be eliminated from the rewritten pattern. We can eliminate these within a disjunction or over the full conjunction of disjunctions. To start, we examine pairs of happens-before relationships within a single disjunction. We then extend our analysis to more complex combinations.

The most obvious case where we can remove a happens-before relationship is when it is equivalent to another relationship in the same disjunction. For example, if our disjunction were $(\$a \rightarrow \$b \vee \$a \rightarrow \$c \vee \$a \rightarrow \$b)$, then clearly either the first or last happens-before relationship can be removed.

A more complex case occurs when one happens-before relationship is a subset of

another. For instance, if we consider the disjunction $(a \rightarrow b \vee a \nrightarrow c \vee b \nrightarrow a)$, it is not immediately clear what we can do. Consider the fact that $a \rightarrow b$ is a subset of $b \nrightarrow a$ since $b \nrightarrow a$ is equivalent to $a \rightarrow b \vee a \parallel b \vee a = b$. Since $a \rightarrow b$ and $b \nrightarrow a$ require the same amount of time to evaluate, and a check for $a \rightarrow b$ is essentially “built-in” to $b \nrightarrow a$, leaving $a \rightarrow b$ in the expression is unnecessary and, in fact, removing it will cause us to perform fewer evaluations.

There is one more case involving pairs of happens-before relationships in a disjunction. In some cases, there may be two pairs that are complements of each other. Because this type of disjunction will always be true, we can completely remove it from the conjunction of disjunctions. One example of this is $(a \rightarrow b \vee a \nrightarrow c \vee a \nrightarrow b)$.

We continue to explore disjunctions, but instead of pairs of happens-before relationships, we will examine interactions among three or more happens-before relationships. If we have multiple relationships satisfying some of the two-relationship scenarios, we can perform the necessary actions multiple times until we have reduced the relationships such that no more scenarios are satisfied. For example, the disjunction $(a \rightarrow b \vee a \rightarrow b \vee b \nrightarrow a)$ could be transformed first to $(a \rightarrow b \vee b \nrightarrow a)$ and then to $(b \nrightarrow a)$. There is however one set of scenarios unique to three or more happens-before relationships. These scenarios involve the transitive property of the happens-before operator for primitive events. The transitive property states that $a \rightarrow b \wedge b \rightarrow c \Rightarrow a \rightarrow c$. The contrapositive, $a \nrightarrow c \Rightarrow a \nrightarrow b \vee b \nrightarrow c$, can also be considered. The property can be extended to any number of happens-before relationships (i.e., $a_1 \rightarrow a_2 \wedge a_2 \rightarrow a_3 \wedge \dots \wedge a_{n-1} \rightarrow a_n \Rightarrow a_1 \rightarrow a_n$). Unfortunately, for disjunctions, we can not use this property to remove any happens-before relationships from the disjunction. For example, in the three-variable case, $a \rightarrow c$ being false does not guarantee that both $a \rightarrow b$ and $b \rightarrow c$ are false, so we must still check both of them. Likewise, if $a \rightarrow b$ and $b \rightarrow c$ are false,

it tells us nothing about $a \rightarrow c$. We will, however, be able to make use of this property when evaluating the conjunction of disjunctions.

After we have reduced each individual disjunction as much as possible, we can attempt to prune over the entire conjunction of disjunctions. Some of the same ideas as for single disjunctions still apply. If two disjunctions in the conjunction are equal (i.e., they contain identical happens-before relationships), then we can remove one of them from the conjunction. If we discover two disjunctions that are complements of each other, then the entire conjunction will always return false and no further processing on the pattern is needed. The only disjunctions we check are those with single happens-before relationships, as the complement of a disjunction containing multiple happens-before relationships is a conjunction and there seems to be no straightforward way to determine equivalence between a conjunction and a disjunction.

When checking for disjunctions that are subsets of other disjunctions, again, for simplicity, we only examine disjunctions with single happens-before relationships. The disjunction that is the larger set is removed from the conjunction as the intersection of the set and subset (i.e., the subset) is what is required to be true to satisfy both disjunctions.

Finally, we examine sets of three or more happens-before relationships across multiple disjunctions, taking advantage of the fact that $a_1 \rightarrow a_2 \wedge a_2 \rightarrow a_3 \wedge \dots \wedge a_{n-1} \rightarrow a_n \Rightarrow a_1 \rightarrow a_n$. Each of these happens-before relationships needs to be present in a separate disjunction (and each such disjunction must contain no other happens-before relationships). We then remove the disjunction containing $a_1 \rightarrow a_n$ as if all the other disjunctions are true (a necessary requirement for satisfying the conjunction), then it will be true. It is not clear, however, that we *should* remove this disjunction. For example, consider the scenario where $a_1 \rightarrow a_2 \wedge a_2 \rightarrow a_3 \wedge \dots \wedge a_{n-2} \rightarrow a_{n-1}$ is true, but $a_1 \rightarrow a_n$ is false. If we had placed

$a_1 \rightarrow a_n$ before the other disjunctions (rather than removing it), we could have avoided $n - 2$ extra comparisons. (We will examine the ordering of disjunctions and happens-before relationships in the following sections.) The cost of removing the disjunction relative to keeping it could be quite high in some situations. Given this, we do not eliminate such disjunctions from the conjunction.

Unfortunately, if patterns are written efficiently from the start, then very few, if any, of these pruning techniques can be applied. They will be of most benefit to users who are unable to specify efficient patterns, and for very complex patterns that are difficult for even experienced users to specify efficiently. It is possible that the process of rewriting patterns could introduce some redundancies, but we have not encountered a pattern in which this is the case. In Section 6.4, we examine some inefficiently-written patterns to verify that the pruning techniques work as expected.

6.2 Static Reordering

The original search algorithm allowed for limited flexibility with respect to the order in which nodes could be evaluated. A parent node could not be evaluated until both its children were evaluated. The use of rewritten patterns and the new search algorithm (as presented in Chapter 5) offers more flexibility. The disjunctions in the rewritten pattern can be evaluated in any order and the happens-before relationships within disjunctions can be re-arranged as well. In this section we address ways to re-order the pattern before beginning the search with the goal of reducing the number of relationships we need to check during the search. Three different reordering techniques, each increasing in complexity, are described.

6.2.1 Brute-Force Technique and Work Estimation

The first technique is a “brute force” method. There are a limited number of ways we can reorder a conjunction of disjunctions. The disjunctions can be ordered within the conjunction in $d!$ different ways (where d is the number of disjunctions), and the happens-before relationships in a disjunction (in position i within the conjunction) can be reordered in $h_i!$ ways, where h_i is the number of happens-before relationships in that disjunction. Overall, there are $d! * \prod_{i=1}^d h_i!$ possible orderings of the conjunction. To determine which ordering is best, we need a way to estimate how many operations will be required to evaluate a particular re-ordered pattern.

Our estimation begins by examining the individual disjunctions (but will also consider the ordering of the disjunctions in the conjunction). It is important to note that we only stop processing a disjunction once we have successfully matched one of the happens-before relationships. As long as we continue to fail to match relationships, we have to continue examining further relationships in the disjunction. If we knew exactly how many times a given relationship in a disjunction succeeded or failed as well as information about the dependencies among the various happens-before relationships, then we could make a reasonable prediction about how many relationships in the disjunction that we would have to examine. Unfortunately, even with this information, we can not be certain about the accuracy of this prediction. There may be dependencies between various relationships that are not evident from the disjunction itself. For example, in the disjunction $a \rightarrow b \vee b \rightarrow c \vee a \rightarrow d$, there may be some correlation between two of the happens-before relationships (say $a \rightarrow b$ and $a \rightarrow d$). Perhaps when $a \rightarrow b$ is false it is more likely that $a \rightarrow d$ is true. If such situations exist, then our calculations will not be as accurate as we would like. Although it may be possible to discover or estimate the impact of such dependencies, we do not attempt to do so in this thesis.

Even when we assume independence among happens-before relationships in a disjunction, the knowledge of how many times a given relationship in a disjunction will succeed or fail is not trivial to discover. In our approach, we take a random sampling of pairs of events that satisfy the variables in the relationship and discover the portion of them that satisfy the relationship. Since we acquire a full list of matches for each variable as part of the pattern search, we multiply this estimated success rate by the product of the number of matches from each variable to determine an estimate of how many matches will succeed (and how many will fail). Based on that information, we estimate, for a specific ordering of an entire disjunction, the expected number of happens-before relationships we need to evaluate before we either succeed or reach the end of the disjunction. By “evaluate”, we mean “examine a single pair of values and determine if they satisfy the given happens-before relationship”. To estimate the number of times we need to evaluate exactly the first i happens-before relationships, we look at the first $i - 1$ happens-before relationships in the disjunction and take the product over the number of pairs that will fail each one and multiply this by the number of pairs that will successfully match the current relationship. Once we compute this information for all values of i , we determine the expected number of happens-before relationships we need to examine for a given ordering. Specifically, the expected number of happens-before relationships we will examine in a single disjunction (over the course of an entire evaluation of a pattern) is

$$\sum_{i=1}^{h-1} [i * l_i * r_i * success_i * \prod_{j=1}^{i-1} (l_j * r_j * (1 - success_j))] \\ + [h * l_h * r_h * \prod_{j=1}^{h-1} (l_j * r_j * (1 - success_j))]$$

where h is the number of happens-before relationships in the disjunction, l_j and r_j are the number of events, respectively, matching the criteria of the left variable/class

and the right variable/class of happens-before relationship j , and $success_j$ is the percentage of these left-right pairs that satisfy the happens-before relationship (we estimate this through random sampling). Note that we treat the case of $i = h$ separately as the h^{th} term is the final happens-before relationship and our success at evaluating it does not effect the cost of evaluating the disjunction.

One important piece of information we neglected to consider above is whether a variable we encounter as part of a happens-before relationship in a disjunction was previously bound or not. (We assumed above that they never are previously bound.) For a more realistic estimate, we track which variables have been already assigned values from disjunctions that appear earlier in the conjunction as well any variables that have been assigned values from earlier happens-before relationships within the current disjunction. If a variable has already been assigned, then for our calculations, we consider it as only having a single matching event in the current position. Specifically, we set l_i , l_j , r_i , and/or r_j (depending on which were previously bound) to 1 in the earlier calculation.

Finally, for each disjunction, we need to calculate the probability that it fails (i.e., $\prod_{i=1}^h (1 - success_i)$, where there are h happens-before relationships in the disjunction and $success_i$ is the probability that each will succeed) and, thus, that we will need to backtrack. (Recall that a disjunction only fails if all happens-before relationships in it fail simultaneously.) Again, we assume independence among the different variables.

Once we have collected this information for each disjunction, we calculate the expected number of evaluations that need to be done over the entire conjunction (in its current ordering) using a similar technique as for the disjunctions. The difference, when evaluating a conjunction, is that we backtrack when encountering the first failure. We need the previously computed information about disjunctions: how many happens-before relationships will be evaluated for each ($evals_i$) and the

failure rates for each disjunction ($failure_i$). Then, we can compute the expected number of happens-before relationships evaluated for a conjunction over the course of an entire pattern evaluation as

$$\sum_{i=1}^{d-1} [i * evals_i * failure_i * \prod_{j=1}^{i-1} (evals_j * (1 - failure_j))] \\ + [d * evals_d * \prod_{j=1}^{d-1} (evals_j * (1 - failure_j))]$$

where d is the number of disjunctions in the conjunction.

Now that we have a formula that estimates, in essence, the amount of work that would need to be done for a given ordering, we can attempt to use the formula to check every permutation of a given pattern, as mentioned earlier, and then choose the permutation that we estimate will involve the least work. Unfortunately, if d is large, or many values of h_i are large, then this is not a practical technique. Our development of this formula, however, is useful in other ways. It has given us insight into some of the complexities that influence the amount of work done to evaluate a pattern and may be helpful when developing other reordering techniques.

6.2.2 Least/Most-Successful-First Technique

For short patterns, computing all permutations, estimating the cost of each, and choosing the least-cost permutation is feasible; however, even a small increase in the size of a pattern will significantly increase the cost of this method. Having other techniques that proceed in a more direct way to order a pattern will be necessary in many cases.

One basic technique we use involves first calculating $l_i * r_i * success_i$ for each happens-before relationship (we assume no previous variable bindings). Then, within each disjunction, we sort the happens-before relationships based on these

values from high to low (since succeeding early in a disjunction will result in fewer evaluations over the entire disjunction). Next, we calculate the expected number of happens-before relationships we will examine in a single disjunction, as in Section 6.2.1, again assuming no previous variable bindings. We then use these final numbers to sort the disjunctions from low to high (since failing early in a conjunction will result in fewer evaluations over the entire conjunction).

The downside of this method is that it ignores any variable bindings done previously in the pattern. In order to make this calculation simple, it was a necessary omission; however, it may result in a sub-optimal ordering.

6.2.3 Variable-Clustering Technique

Our variable-clustering technique builds on the technique from Section 6.2.2; however, it tries to take into consideration the benefits that an earlier variable binding has on the number of evaluations needed. We also leverage some of the analysis done in Section 6.2.1 when creating the function that estimated the amount of work required for a given pattern ordering.

Once a variable has been assigned a value, all other happens-before relationships involving that variable become less expensive to evaluate as the value for l_i , r_i , or both, becomes 1. Our goal is to evaluate as few unassigned variables as possible early in the pattern since the costs of evaluations early in the pattern affect the overall cost of the pattern significantly more than the costs of evaluations later in the pattern.

One step towards this goal involves determining which variable appears most in the pattern and placing a disjunction containing that variable at the start of the conjunction. For example, if variable v_1 occurs most frequently, we would examine disjunctions containing v_1 for consideration for placement at the start of

the pattern. Within the disjunctions containing v_1 , we would like to place the one having the fewest unassigned variables first. For example, we would favour the disjunction $(v_1 \rightarrow v_2)$ over $(v_1 \rightarrow v_3 \vee v_2 \rightarrow v_4)$ as the first contains one unassigned variable, not including v_1 , whereas the second contains three. To break ties (for example, between disjunctions $(v_1 \rightarrow v_2)$ and $(v_1 \rightarrow v_3)$) we can consider which of the unassigned variables occurs most frequently in the pattern, and choose that one.

Once we have decided on this first disjunction, we order the happens-before relationships in the disjunction by placing the relationships containing the fewest unassigned variables first and breaking ties as before. Once this is done, the unassigned variables in this disjunction are now considered “assigned” for purposes of choosing the second disjunction and we continue with the second iteration of the algorithm. Ideally, we would like to find a second disjunction that contains only assigned variables; however, if that is not possible, we again choose the disjunction with the fewest unassigned variables and break ties as before. Again, we may increase the size of our “assigned” set and continue adding disjunctions, after reordering their happens-before relationships, to the new reordered conjunction until all have been added.

Although this technique may not produce an optimal ordering, it does consider more factors than the technique in Section 6.2.2 and will likely produce a better ordering.

The three reordering techniques are evaluated in Section 6.4.

6.3 Processing Universally-Quantified Variables

The changes mentioned in this section involve modifying the algorithm that evaluates a pattern, rather than the pattern itself.

Evaluating expressions involving universally-quantified variables can often be time-consuming. Consider, for example, the simple pattern $a \rightarrow *b$. For each value of a , we have to examine values of $*b$ until we find one that does not satisfy the happens-before relationship. When we choose a new value for a , we repeat this process, starting again with the first value of $*b$. In many cases, though, certain values of $*b$ will be far more likely not to satisfy the relationship than others. For example, as we evaluate values of a that occur “late” in our event set, values of $*b$ that occur “early” will be more likely not to satisfy the relationship, on average.

We would like to change the algorithm so that values of $*b$ less likely to satisfy the relationship are tested first and values of $*b$ that are more likely to satisfy the relationship are tested later. One way we can approximate this is through a least-recently-used ordering. Our assumption is that values for $*b$ that have caused the relationship to fail most recently will be most likely cause it to fail in the near future.

We evaluate this approach in Section 6.4 against patterns containing universal quantifiers.

6.4 Performance Evaluation

All the techniques presented in sections 6.1 through 6.3 are evaluated in this section. We first examine each technique separately on a small number of specific test cases, and then, using the best techniques in conjunction, we run the full pattern-based test suite (as described in Chapter 4) to measure the overall change in runtime.

All the tests were run on a Core 2 Quad 3.0 GHz CPU with 6 GB of RAM under the Fedora Core 8 distribution of Linux. The underlying database used was MySQL version 14.12. Each time in the table is the average over five test runs (except for

one test which took nearly 10 hours) and the range of results for each set of test runs was never greater than 7% of the average value over those test runs.

To run each test in this section, we first followed the steps described in Section 4.6. In all cases, after setting the appropriate flags (as described below), we ran the tests using the “Searcher” command from Section 4.6. We set argument \$1 to the name we gave the project into which we imported the data set, \$2 to the name we gave the data set, \$3 to the name of the pattern file in which we typed our pattern, \$4 to the name of the pattern we want to search on (e.g., “ConSend”), and \$5 to “flat”.

To turn on pruning, the flag PRUNING is set to true in FindFlattenedMatch.java, located in poet.core.pattern. To turn on a specific reordering technique, the flag REORDERING is set to the appropriate value as described in FindFlattenedMatch.java, located in poet.core.pattern. Enabling the new technique for universally-quantified variables requires us to set the TRACK_FA flag to true in Factor.java, located in poet.core.pattern.

In our first tests, we evaluate the pruning techniques of Section 6.1. As mentioned in that section, the pruning techniques are most useful for inefficiently specified patterns. Thus, we modify some of the existing patterns by adding unnecessary terms and compare the performance of the pattern search with and without pruning on these patterns. The first pattern we modify is the ConnectionEstablished pattern, which operates on the TCP data set, and we write it instead as ConnectionEstablished2 as shown in Figure 6.1. (We also create FirstConnectionEstablished2 and LastConnectionEstablished2 patterns to search based on this ConnectionEstablished2 pattern.) It is feasible that a user might write a pattern like ConnectionEstablished2 without realizing it is equivalent to the ConnectionEstablished pattern and pruning can help to correct these user-induced inefficiencies. In our first four test cases, shown in Table 6.1, we search with patterns FirstConnectionEstablished2

```

StartConnect := ["Server", "Accept", ""] .
               ["Listen Socket", "Accept_stream", ""];
DoneConnect := ["Server", "Accept_done", ""] .
               ["Listen Socket", "Accept_done_stream", ""];
StartConnect $sc, *sc_all;
DoneConnect $dc;

ConnectionEstablished := ($sc --> $dc) &
                        (($sc !--> *sc_all) |
                         (*sc_all !--> $dc));
ConnectionEstablished2 := ($sc --> $dc) &
                        (((($sc --> *sc_all) &
                          (*sc_all !--> $dc)) |
                          ($sc !--> *sc_all)));
ConnectionEstablished $ce_first, *ce_all;
ConnectionEstablished2 $ce_first, *ce_all2;

FirstConnectionEstablished := *ce_all !-->
                             ConnectionEstablished;
FirstConnectionEstablished2 := *ce_all2 !-->
                             ConnectionEstablished2;

LastConnectionEstablished := ConnectionEstablished2 !-->
                             *ce_all;
LastConnectionEstablished2 := ConnectionEstablished2 !-->
                             *ce_all2;

```

Figure 6.1: Inefficiencies Added, Example 1

and LastConnectionEstablished2, with both pruning turned on and off, and, as a benchmark, compared their runtimes against searching for FirstConnectionEstablished and LastConnectionEstablished, again with and without pruning.

For our fifth and sixth pruning test cases, we write a SendRecv2 pattern, which is a modified version of the SendRecv pattern. It operates on the PVM Life data set described in Chapter 4. The inefficiency we added is an unnecessary happens-before relationship as shown in Figure 6.2. Again, we evaluate the SendRecv and SendRecv2 patterns with and without pruning turned on.

```

Send := ["", "send", ""];
Recv := ["", "recv", ""];
ANY := ["", "", ""];

SendRecv := (Send -(ANY)-> Recv);
SendRecv2 := (Send -(ANY)-> Recv) & (Send --> Recv);

```

Figure 6.2: Inefficiencies Added, Example 2

Pattern	Average Time Elapsed		Timestamp Comp.	
	No Pruning	Pruning	No Pruning	Pruning
FirstConEstab	20 ms	23 ms	644	644
FirstConEstab2	73 ms	28 ms	8192	948
LastConEstab	20 ms	24 ms	804	804
LastConEstab2	65 ms	38 ms	4676	1304
SendRecv	299 s	299 s	$1.263 * 10^9$	$1.263 * 10^9$
SendRecv2	328 s	298 s	$1.326 * 10^9$	$1.263 * 10^9$

Table 6.1: Pruning Performance Analysis

The results in Table 6.1 demonstrate that pruning improved the performance of the patterns that contained unnecessary terms. Although pruning was not always able to remove all unnecessary terms (only in the final two test cases was it able to), pruning still reasonably decreased the runtime compared to searching with the non-pruned pattern.

Our next tests evaluate the reordering techniques from Section 6.2. We examine the performance of all three reordering techniques, first, on two small patterns, and then, on four larger patterns. The tests are taken from the suite of tests presented in Chapter 4. Cases 9.1 and 9.2 use patterns ConSendNew and ConSendNew2, respectively, as shown in Figure 6.3. These are variations on Case 9 (ConSend).

```

Send := ["", "send", ""];
ConSend := (Send || Send || Send || Send || Send ||
            Send || Send || Send || Send);
ConSendNew := (Send || Send || Send || Send || Send ||
              (Send || Send) || Send || Send);
ConSendNew2 := ((Send || Send) || (Send || Send)) ||
              ((Send || Send || Send) || (Send || Send));
ConSendExplicit := (Send || (Send || (Send || (Send ||
              (Send || (Send || (Send || (Send ||
              Send)))))))));

```

Figure 6.3: Test Cases 9, 9.1, and 9.2

Case	Average Time Elapsed / Timestamp Comparisons			
	No Reordering	Technique #1	Technique #2	Technique #3
4	189 ms / 144,776	680 ms / 144,776	243 ms / 161,016	215 ms / 144,776
5	87 ms / 59,400	72 ms / 31,380	89 ms / 59,400	72 ms / 31,380
8	4529 ms / 3732	N/A	4548 ms / 3732	4561 ms / 1968
9	75 s / $0.508 * 10^9$	N/A	75 s / $0.508 * 10^9$	74 s / $0.508 * 10^9$
9.1	108 s / $0.903 * 10^9$	N/A	108 s / $0.903 * 10^9$	70 s / $0.500 * 10^9$
9.2	32,633 s / $285 * 10^9$	N/A	32,602 s / $285 * 10^9$	68 s / $0.490 * 10^9$

Table 6.2: Reordering Performance Analysis

Some observations can be made about these results, shown in Table 6.2. First, the reordering technique that examines all permutations is not very useful in practice. For case 4, it was too slow, and for cases 8 through 9.2, there was not enough memory available to compute the permutations. The least/most-successful-first technique was also not very useful. It was never better than the third technique (and was often much worse), and in some cases was even worse than using no reordering at all. Finally, the third “variable-clustering” technique seemed to be useful in all cases we examined.

Cases 9.1 and 9.2 were created and tested to demonstrate that, for some patterns, the original, non-reordered rewritten pattern can be optimal, but that if slight changes are made to the original pattern, it can become non-optimal. An example of a pattern that has a natural optimal order is the pattern ConSend. We illustrate

how it is parsed by the pattern-search algorithm through explicit brackets, as shown in the pattern `ConSendExplicit` (Figure 6.3). When it is parsed this way, the pattern is rewritten in a very similar way to the “variable clustering” technique. Only one new variable is introduced at each level of the parse tree, and all relationships between that variable and each of the previously parsed variables are immediately added to the rewritten pattern. This means that we explore as few new variables as possible as we process the rewritten pattern.

If we instead force the pattern to be parsed in a different order, the pattern search, without optimization, shows a significant degradation in speed. This is illustrated to a small extent in `ConSendNew` (shown in Figure 6.3), where we only modify the parse order slightly, and much more significantly in `ConSendNew2` (again, shown in Figure 6.3), where we aim to make the left and right sides of the parse tree as equal in height as possible. In both these cases, reordering is necessary to maintain the original speed of the pattern evaluation, as shown in the test results presented in Table 6.2. Evaluating cases 9.1 and 9.2 using the optimized convex-closure-based pattern search from Chapter 4 resulted in a similar degradation. Case 9.1 took 2.6 times as long as case 9 and case 9.2 took 350 times as long as case 9. These additional results show that the degradation in speed is not related specifically to the rewriting algorithm.

Next, we examine how the new method for processing universally-quantified variables from Section 6.3 affects the runtime of the pattern search by looking at the two most time-consuming patterns from our set of eleven tests. Because these two patterns are simple, but contain universal quantifiers, it is reasonable to assume that their slowness is mostly due to how universally-quantified variables are processed. Table 6.3 provides an evaluation of the existing versus new technique for processing these types of variables.

Case	Average Time Elapsed		Timestamp Comp.	
	Old	New	Old	New
10	281 s	4 s	$1.234 * 10^9$	$0.015 * 10^9$
11	299 s	10 s	$1.263 * 10^9$	$0.033 * 10^9$

Table 6.3: Pruning Performance Analysis

It is clear from the results that this new way of processing universally-quantified variables has a very significant effect on the speed of the pattern search. Both patterns are now evaluated more than 900 times faster compared to the convex-closure approach described in Chapter 4.

Finally, we examine the combined performance of both the variable-clustering reordering technique and universally-quantified-variable optimizations on our entire test suite of eleven patterns. The values in the “Unoptimized” columns are obtained from executing the rewriting approach with the given patterns, but using no optimizations described in this chapter. The results of these tests are shown in Table 6.4.

Case	Average Time Elapsed		Pattern Size	Timestamp Comp.	
	Unoptimized	Optimized		Unoptimized	Optimized
1	20 ms	145 ms	59	644	644
2	20 ms	144 ms	59	804	804
3	42 ms	53 ms	4	7672	7672
4	189 ms	121 ms	10	144,776	86,312
5	87 ms	68 ms	2	59,400	8940
6	12,001 ms	12,101 ms	2	37,464,278	37,464,278
7	8236 ms	8356 ms	5	13,349,356	13,349,356
8	4529 ms	4525 ms	10	3732	1968
9	75 s	74 s	72	$0.508 * 10^9$	$0.508 * 10^9$
10	281 s	4 s	3	$1.234 * 10^9$	15,362,041
11	299 s	10 s	3	$1.263 * 10^9$	33,322,558

Table 6.4: Full Performance Analysis

The combination of these two techniques provides a significant improvement in runtime for several patterns, but results in a slightly longer evaluation time in other

cases. The number of timestamp comparisons follows a similar trend. As mentioned previously, in many cases, the specification and parsing of patterns naturally causes variables to be clustered together, and so the reordering technique makes no improvement in those cases. Furthermore, the overhead from the reordering technique causes some decrease in the speed of the evaluation of such patterns, especially when they are large.

Certainly, the improvements in the more complex and more time-consuming patterns makes reordering worthwhile; however, developing a means to decide when reordering should be used would allow evaluations of the fast, less complex patterns to maintain their speed.

Chapter 7

Closing Remarks

MAKING SENSE of large, complex data sets is difficult, no matter what the domain. It is particularly challenging when working with data collected from a distributed system that potentially contains numerous concurrent entities and where the data is ordered solely by communications. As distributed and multi-CPU systems become more prevalent and more necessary, tools that can help us debug, visualize, and understand these systems, as well as assist us in making them more dependable and help us improve their performance, are crucial. This thesis addressed these challenges and provided solutions that resulted in a more consistent framework, a more expressive pattern language, and a more efficient pattern search.

7.1 Conclusions

Our primary goal in this thesis was to develop a search facility for partial-order event data collected from a distributed system that would be efficient, allow expressive search patterns to be specified, and provide consistent, well-defined results. This thesis provides several significant contributions towards these goals.

1. We developed a provably-complete framework for evaluating precedence between event sets, remedying difficulties with previously used frameworks. These frameworks were incomplete and did not fully address the unique issues of evaluating event sets as compared to single events.
2. We added general features to the language used to specify search patterns and demonstrated the increased expressiveness that these features provide.
3. We greatly improved the runtime of the algorithm that builds convex events and showed how any two convex events can be combined without having to rebuild the convex event by extracting the individual events from each. We also demonstrated, through empirical results, the improved runtime that these improvements provide.
4. We demonstrated how a search pattern can be rewritten into a less complex and more flexible form. The rewritten form of a pattern can be evaluated more quickly, as demonstrated through empirical analysis.
5. Finally, we showed how various optimizations to the rewritten pattern and the processing algorithm can further improve the runtime of the pattern search and again demonstrated the potential of these optimizations through test cases.

The work presented here should serve as a solid base for those pursuing future work in this area.

7.2 Future Work

There are several opportunities for future work in this area and most involve exploration of techniques that will further improve the runtime of the search algorithm.

7.2.1 Dynamic Reordering of Rewritten Patterns

Although static reordering, as described in Chapter 6, chooses an ordering for the elements in a rewritten pattern and uses it for the duration of the pattern search, dynamic reordering involves changing the ordering during the pattern search based on metrics we have collected as part of the the search thus far. There are a few observations we can make about this type of reordering.

First, we can easily reorder parts of the pattern that do not currently have events assigned to them (i.e., if we explore the pattern from left to right, then elements to the right of our current location can be reordered). We are permitted to do this as we have either never reached these parts of the pattern, or our last action was to backtrack from these elements to an earlier pattern element. A second observation is that any pattern elements that do have events assigned to them (i.e., elements to the left of our current position) would be difficult to reorder dynamically and doing so could result in losing previous work. We could also end up reaching the end of a pattern having not satisfied all the elements (if we moved an element dynamically from the right of our current location to the left of our current location at some point during the search).

Reordering of events in this second scenario could be difficult to implement. If many of these types of reorderings take place over the duration of a single search, the problem becomes more complex. Keeping a complex accounting of which matches have been explored and which pattern elements have been satisfied thus far seems necessary if we wish to allow for full reordering of the pattern at any time.

7.2.2 Mixed-Strategy Approach

As the numbers of variables and relationships in a pattern increase, so does the complexity of the rewritten pattern. As we reach the top of the pattern parse tree,

the number of variables that have been propagated up from each side can be quite large and the number of relationships we need to add to the rewritten pattern is proportional to the product of the numbers of unique variables in each subtree. In contrast, the complexity of the convex-closure method does not grow significantly at each level of the tree. In fact, increasing the number of input events can reduce the complexity of computing a convex closure if events are confined to a small number of traces or if they are not far apart from each other.

A straightforward approach to handle this situation could involve examining a pattern and, based on its complexity, choose *either* the rewriting or convex-closure method. Another, more complex, approach would be to use a “mixed-strategy” approach. This strategy would involve evaluating certain subtrees using the rewriting method, but then passing the matches of those subtrees up to a parent that then utilizes the convex-closure method. Determining in which situations, if any, such a technique should be used would be the most difficult challenge of this approach.

7.2.3 Parallelization of Search Algorithm

The flexibility of rewritten patterns could allow us to execute parts of the pattern in parallel. Some analysis would need to be done on which parts could be run in parallel. For example, two pattern components that share a variable that was not previously assigned a value could not be run in parallel as they could potentially pick two different values for the same variable. Also, we would need to evaluate the effectiveness of this approach. If, for example, we process a pattern from left to right, but evaluate two components in parallel, we could often end up backtracking before reaching the second component. In that case, the work done evaluating the second component would be wasted. Some exploration into evaluating a pattern in

a non-linear way that favours parallelism could be explored. We should also weigh the value of other opportunities for parallelism (such as computing metrics while the search is ongoing) as they might provide a more significant benefit.

7.2.4 Deeper Exploration of Reordering Metrics

Although several reordering strategies were explored in this thesis, the problem of finding the “best ordering” of a rewritten expression is complex. As mentioned in Chapter 6, finding the optimal strategy for evaluating a database query is still an open problem and seems similar to our problem. Developing a larger selection of useful patterns could provide some additional case studies that, when examined, could provide some greater insight into additional reordering strategies. These additional patterns would also be helpful in evaluating other improvements to the pattern search.

7.2.5 Lower Bound on Convex-Closure Algorithm

A variety of improvements have been made to the convex-closure algorithm; however, we are still uncertain if there is potential for further improvement. To this end, computing a lower bound on the computation of a convex closure would be helpful. Unfortunately, there are many different variables to consider when computing such a bound, such as the number of traces and events in the data set and the number of events and traces involved in the resulting closure. Discovering a lower bound based on all of these variables could be complicated.

7.2.6 Online Pattern Search

The pattern search is designed to be run offline. The search assumes that the entire data set is accessible at the start of the search and that new data is not added during the search. Building an online version of the pattern search involves keeping account of partial matches to the pattern set as additional events may arrive that complete these matches. The challenge is that often not all of these partial matches can be stored. Potential solutions involve limiting the span of the match to a pattern (i.e., only return matches that fall within a small portion of the data set) or returning only a subset of the matches, but allowing such matches to span the full data set.

References

- [1] Blue gene home. <http://www.ibm.com/systems/deepcomputing/bluegene/>. 6
- [2] Eclipse.org home. <http://www.eclipse.org>. 8, 68
- [3] HSQLDB home. <http://www.hsqldb.org>. 68
- [4] mySQL home. <http://www.mysql.com>. 68
- [5] TPTP home. <http://www.eclipse.org/tptp>. 69
- [6] M. Autili, P. Inverardi, and P. Pelliccione. A scenario based notation for specifying temporal properties. In *SCESM '06: Proceedings of the 2006 International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 21–28, New York, NY, USA, 2006. ACM. 21
- [7] Twan Basten, Thomas Kunz, James P. Black, Michael H. Coffin, and David Taylor. Vector time and causality among abstract events in distributed computations. *Distributed Computing*, 11(1):21–39, December 1997. 2, 17, 22, 28, 30
- [8] Peter C. Bates and Jack C. Wileden. Edl: A basis for distributed system debugging tools. In *Proceedings of the Fifteenth Hawaii International Conference on System Sciences*, pages 86–93, 1982. 14

- [9] Dwight Bedassé. An efficient computation of convex closure on abstract events. Master's thesis, University of Waterloo, 2004. 2, 66, 68, 84
- [10] Peter Buhr. Understanding control flow with concurrent programming using $\mu\text{C}++$ (draft). 2008. 71
- [11] Peter A. Buhr, Glen Ditchfield, Richard A. Strooboscher, B. M. Younger, and C. Robert Zarnke. $\mu\text{C}++$: Concurrency in the object-oriented language C++. *Software - Practice and Experience*, 22(2):137–172, 1992. 71
- [12] Surajit Chaudhuri. An overview of query optimization in relational systems. In *PODS '98: Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 34–43, New York, NY, USA, 1998. ACM. 21, 111
- [13] Wing Hong Cheung. *Process and Event Abstraction for Debugging Distributed Programs*. PhD thesis, University of Waterloo, 1989. 1, 2, 37
- [14] Jong-Deok Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, Apr 2001. 7
- [15] Michael Coffin and David Taylor. Integrating real-time and partial-order information in event-data displays. In *Proceedings of the 1994 CAS Conference*, pages 157–165, November 1994. 1
- [16] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. Pearson Education, Ltd., 3rd edition, 2001. 6

- [17] Paul S. Dodd and China V. Ravishankar. Monitoring and debugging distributed real-time programs. *Software - Practice and Experience*, 22(10):863–877, 1992. 1
- [18] James Donald and Margaret Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 78–88, Washington, DC, USA, 2006. IEEE Computer Society. 6
- [19] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st International Conference on Software Engineering*, pages 411–420, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press. 21
- [20] Greg Eisenhauer, Weiming Gu, Eileen Kraemer, Karsten Schwan, and John Stasko. Online displays of parallel programs: Problems and solutions. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '97*, pages 11–20, Las Vegas, NV, 1997. 1
- [21] Colin Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, February 1988. 11
- [22] Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991. 11
- [23] Colin Fidge. Fundamentals of distributed system observation. *IEEE Softw.*, 13(6):77–83, 1996. 6
- [24] Mark Fox. Event-predicate detection in the monitoring of distributed applications. Master’s thesis, University of Waterloo, 1998. 2, 6, 14, 15

- [25] Rahul Garg, Vijay K. Garg, and Yogish Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing*, pages 269–277, New York, NY, USA, 2006. ACM. 21
- [26] Vijay K. Garg and Chakarat Skawratananond. Timestamping messages in synchronous computations. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, Washington, DC, USA, 2002. IEEE Computer Society. 14
- [27] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. In *Global States and Time in Distributed Systems*. IEEE Computer Society Press, 1994. 21
- [28] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005. 6
- [29] Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, January 1988. 14, 15, 37
- [30] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991. 1
- [31] Rajneesh Hegde and Kamal Jain. The hardness of approximating poset dimension. In *Proceedings of the European Conference on Combinatorics, Graph Theory and Applications*, pages 435–443. Elsevier, 2007. 14
- [32] Yannis E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996. 21

- [33] Christian Jaekl. Event-predicate detection in the debugging of distributed applications. Master's thesis, University of Waterloo, 1996. 2, 14, 15, 17, 19, 22, 37
- [34] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *International Conference on Software Engineering*, pages 360–370, 1997. 1
- [35] T. Kunz. Automatic support for understanding complex behaviour. In *Proceedings of the International Workshop on Network and Systems Management*, pages 125–132, 1995. 1
- [36] Thomas Kunz. *Abstract Behaviour of Distributed Executions with Applications to Visualization*. PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany, 1994. 1, 2, 17, 37
- [37] Thomas Kunz, James P. Black, David Taylor, and Twan Basten. POET: Target-system independent visualizations of complex distributed-application executions. *Computer Journal*, 40(8):499–512, 1997. 7, 8, 17, 69
- [38] Thomas Kunz and David Taylor. Visualizing PVM executions. In *Proceedings of the 3rd PVM Users' Group Meeting*, Pittsburgh, 1995. 1, 71
- [39] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. 7
- [40] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36(4):471–482, 1987. 7
- [41] Keith Marzullo, Robert Cooper, Mark D. Wood, and Kenneth P. Birman. Tools for distributed application management. *Computer*, 24(8):42–51, 1991.

- [42] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, December 1988. 11
- [43] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *Eighth annual conference on Distributed Computing Systems*, pages 316–323, 1998. 14
- [44] N. Mittal, A. Sen, V. Garg, and R. Atreya. Finding satisfying global states: All for one and one for all. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, 2004. 21
- [45] Matthew Nichols and David Taylor. A mechanism for visualizing TCP-socket interactions. In *Proceedings of the 2005 CAS Conference*, pages 212–224, October 2005. 1, 11
- [46] Matthew Nichols and David Taylor. A faster closure algorithm for pattern matching in partial-order event data. In *Proceedings of the 2007 International Conference on Parallel and Distributed Systems*, December 2007. 2
- [47] Matthew Nichols and David Taylor. Pattern rewriting for efficient search in partial-order event data. In *Proceedings of the 2007 CAS Conference*, pages 58–70, October 2007. 2
- [48] Doron A. Peled. *Software Reliability Methods*. Springer-Verlag, 2001. 1
- [49] Michiel Ronsse and Koen De Bosschere. Replay: A fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999. 7
- [50] Paul E. Slauenwhite. Pattern matching in generic event data. Master’s essay, University of Waterloo, 2007. 2, 39, 45, 69

- [51] Jim A. Summers. Precedence-preserving abstraction for distributed debugging. Master's thesis, University of Waterloo, 1992. 2, 37
- [52] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005. 1
- [53] Andrew Tanenbaum. *Distributed Operating Systems*. Prentice-Hall, Inc., 1995.
- [54] Ashis Tarafdar and Vijay K. Garg. Addressing false causality while detecting predicates in distributed programs. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, pages 94–101, Amsterdam, The Netherlands, 1998. 21
- [55] David Taylor. A prototype debugger for hermes. In *Proceedings of the 1992 CAS Conference*, volume 1, pages 29–42, November 1992. 69
- [56] David Taylor. The POET prototype: Structure and operation. January 1999. 8, 69
- [57] David Taylor. Scrolling partially ordered event displays. *Journal of Parallel and Distributed Computing*, 65(5):643–653, May 2005. 10
- [58] David Taylor and Ping Xie. Specifying and locating hierarchical patterns in event data. In *Proceedings of the 2004 CAS Conference*, pages 81–95, October 2004. 2, 18, 19, 22, 28, 37, 52, 53, 54, 56
- [59] Paul A. S. Ward. An offline algorithm for dimension-bound analysis. In *International Conference on Parallel Processing*, pages 128–, 1999. 14
- [60] Paul A. S. Ward, Tao Huang, and David J. Taylor. Clustering strategies for cluster timestamps. In *Proceedings of the 2004 International Conference on Parallel Processing*, pages 73–81. IEEE Computer Society, 2004. 14

- [61] Paul A. S. Ward and David J. Taylor. A hierarchical cluster algorithm for dynamic, centralized timestamps. In *ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems*, Washington, DC, USA, 2001. IEEE Computer Society. 14
- [62] Paul A.S. Ward. *A Scalable Partial-Order Data Structure for Distributed-System Observation*. PhD thesis, University of Waterloo, 2001. 1, 14
- [63] Paul A.S. Ward and Dwight Bedassé. Fast convex closure for efficient predicate detection. In *Proceedings of Euro-Par 2005 Parallel Processing: 11th International Euro-Par Conference*, pages 30–39, August 2005. 2, 66, 68, 84
- [64] Ping Xie. Convex-event based offline event-predicate detection. Master’s thesis, University of Waterloo, 2003. 2, 17, 18, 19, 37, 54
- [65] Jian Yu, Tan Phan Manh, Jun Han, Yan Jin, Yanbo Han, and Jianwu Wang. Pattern based property specification and verification for service composition. In *Web Information Systems (WISE) 2006*, pages 156–168, 2006. 21